

第1章 多维系统下单点登录深入详解

学习目标

- 目标1：了解淘宝天猫的单点登录系统架构
- 目标2：掌握单点登录的主流实现设计方案
- 目标3：掌握单点登录的主流技术实现方案
- 目标4：深入掌握基于SAML的单点登录技术方案
- 目标5：深入掌握基于OAuth2的单点登录技术方案
- 目标6：完成基于Cookie跨域与分布式Session的技术实践
- 目标7：完成基于Token增强的OAuth2微服务技术实践
- 目标8：完成基于JWT扩展的OAuth2微服务技术实践

1. 从淘宝天猫的单点登录说起

1.1 SSO单点登录

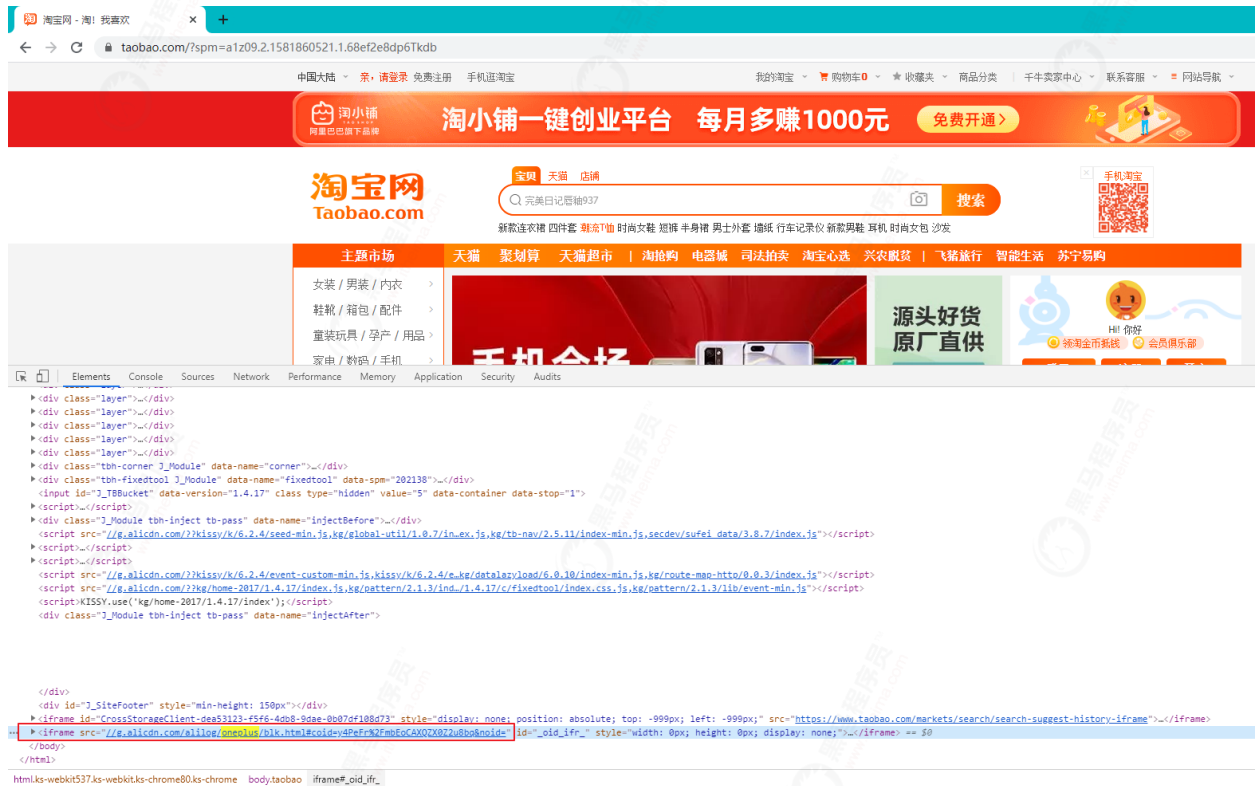
- 概述 随着互联网大数据不断发展，应用服务的不断增多，单点登录越来越能够凸显其作用。单点登录SSO(Single Sign On)，顾名思义就是单个节点登录，全局使用。是目前最为流行的统一登录解决方案。
- 为什么使用？

目的就是为了快速实现用户认证，统一管理用户信息，避免重复维护用户数据；分离用户与业务数据，让业务服务专注于业务功能的实现，让用户中心服务统一认证，减少频繁认证次数，同时保障数据的安全性。
- 应用场景
 - 内部的服务统一认证与授权，比如电商网站，内部的用户服务、订单服务、库存服务、资金服务等，以用户服务作为认证服务中心，实现统一认证与授权。
 - 外部的第三方登录认证与授权，比如登录某个论坛网站，可以采用FaceBook或者Google账号进行登录。
 - 云服务应用，比如使用阿里云的消息推送服务，但不想创建和管理用户，就可以采用基于SAML协议实现SSO单点登录。

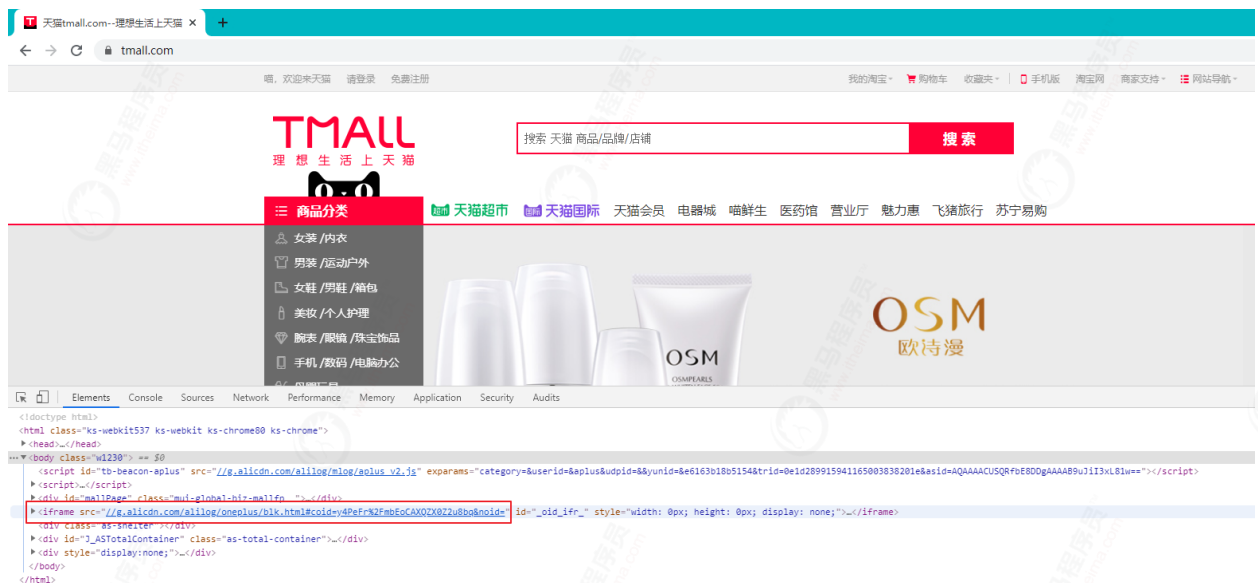
1.2 淘宝天猫登录场景解析

访问淘宝网站，登录之后，再访问天猫网站，你会发现，天猫也是处于登录状态，那么具体是如何实现的？

- 登录技术方案分析 淘宝登录：



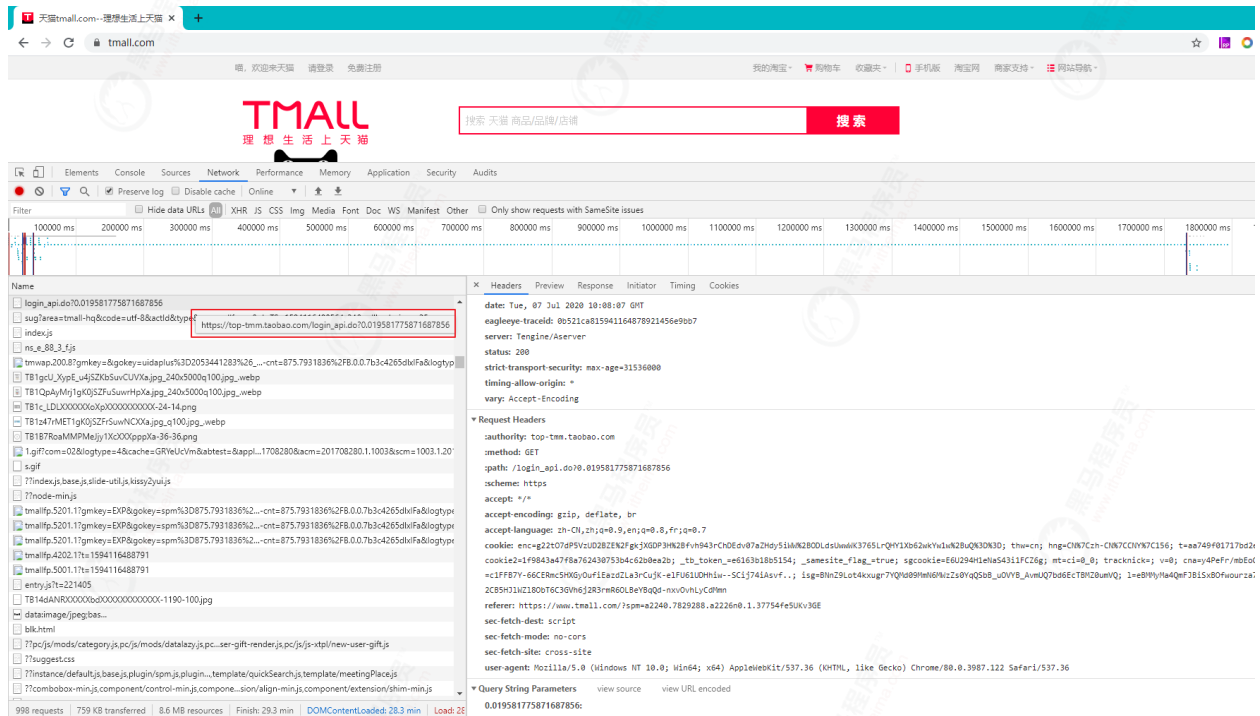
天猫登录:



目前整个登录体系是以淘宝作为中心，天猫通过淘宝作鉴权登录。整个鉴权体系是采用跨域cookie + 分布式session作为解决方案:

淘宝是如何解决Cookie跨域问题，目前淘宝是采用如下方案做处理:

通过内嵌iframe，访问统一域名，实现Cookie信息共享，如果禁用Cookie，你会发现无法正常登录；同时利用静态资源不受同源策略的限制，通过JSONP跨域方式来获取用户的登录状态。



Response会返回Token信息:

```
var userCookie={
  dnk: '', _nk: '', _l_g: '', ck1: '', tracknick: '', mt: 'ci=0_0', l: 'eBMMYMa4QmFJBq7pB05aourza77T3Id
  b4sPzaNbMiInca6BP03JuhNQqw5H95dtjgtC3xetz2m1B9dLHR3fRwxDDBTJbWmu-
  exv0.', uc1: '', t: 'aa749f01717bd2e29ccacc35701ebef7', unb: '', cna: 'y4PeFr/mbEoCAXQZX0Z2u8bq', _t
  b_token_: 'e6163b18b5154', version: '4.0.0'};
window.TB && TB.Global && TB.Global.run &&
TB.Global.run();
```

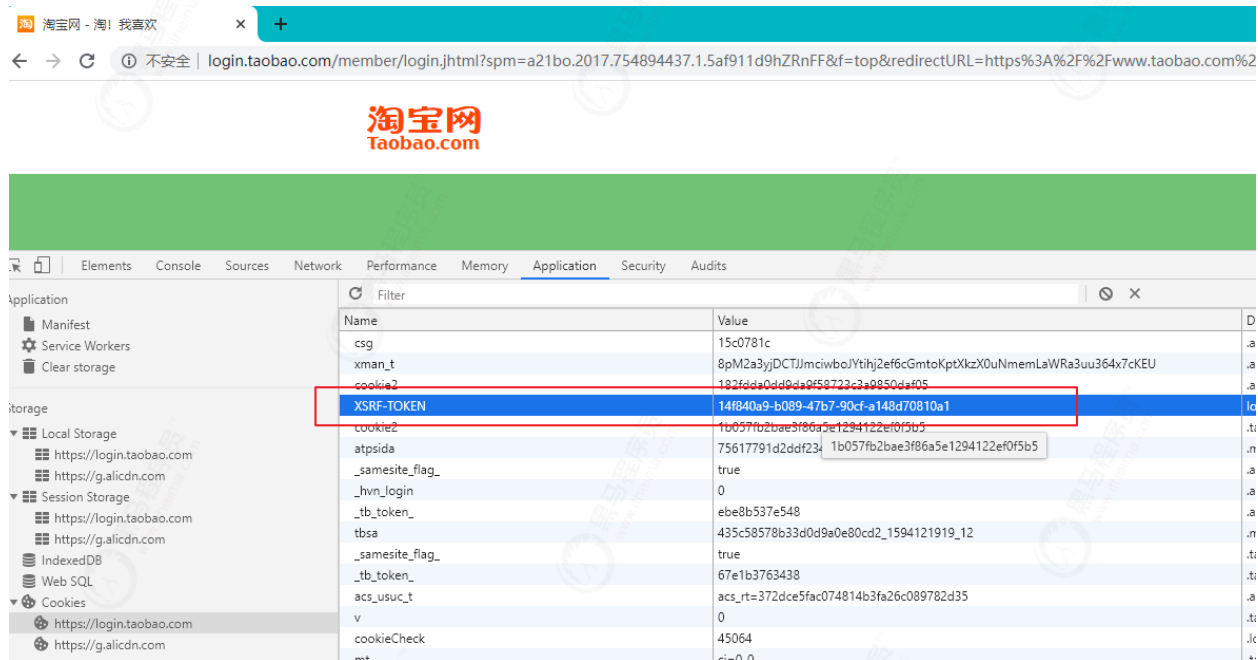
淘宝是如何解决分布式Session管理问题呢? 为了解决此问题, 淘宝专门推出两个重要产品:

第一个是tb session, 基于Tair缓存体系实现的共享Session; 另一个是pass cookie, 解决不同域名之间Cookie同步的问题, 上述的登录鉴权Cookie信息就是通过pass cookie实现的统一管理。

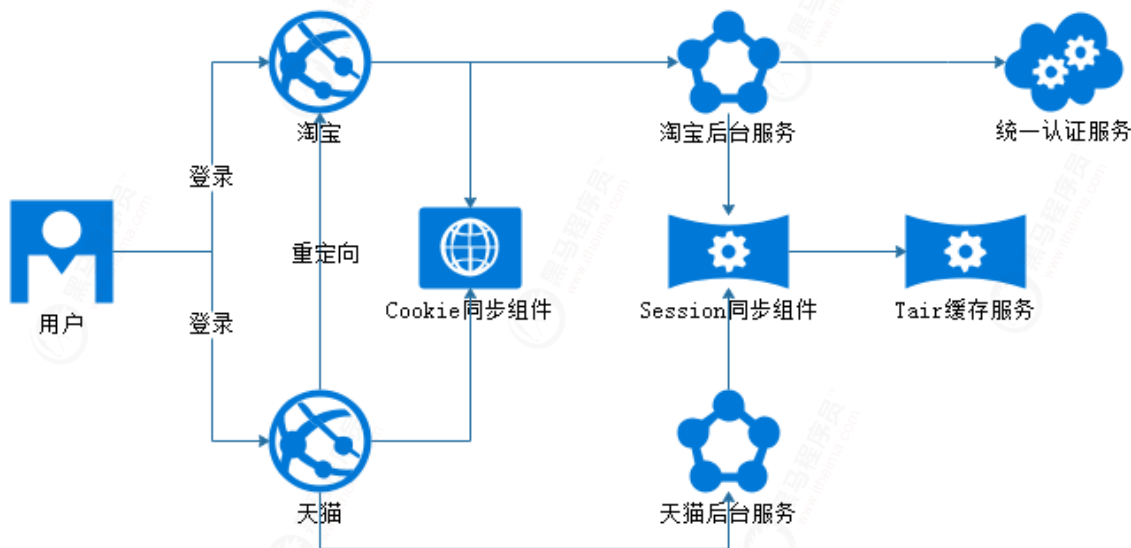
淘宝是如何防范Session劫持?

CSRF/XSRF 攻击的原理, 就是利用浏览器对嵌入资源不做限制的行为进行跨站请求伪造攻击, 比如 `<script>` 等标签。

淘宝是生成一个随机的Token给客户端, 然后提交时在服务端进行校验, 由于Token是不断变化, 并且具有私密性, 只内嵌到当前的用户页面中, 这样就可以防止CSRF的攻击, 保护资源。

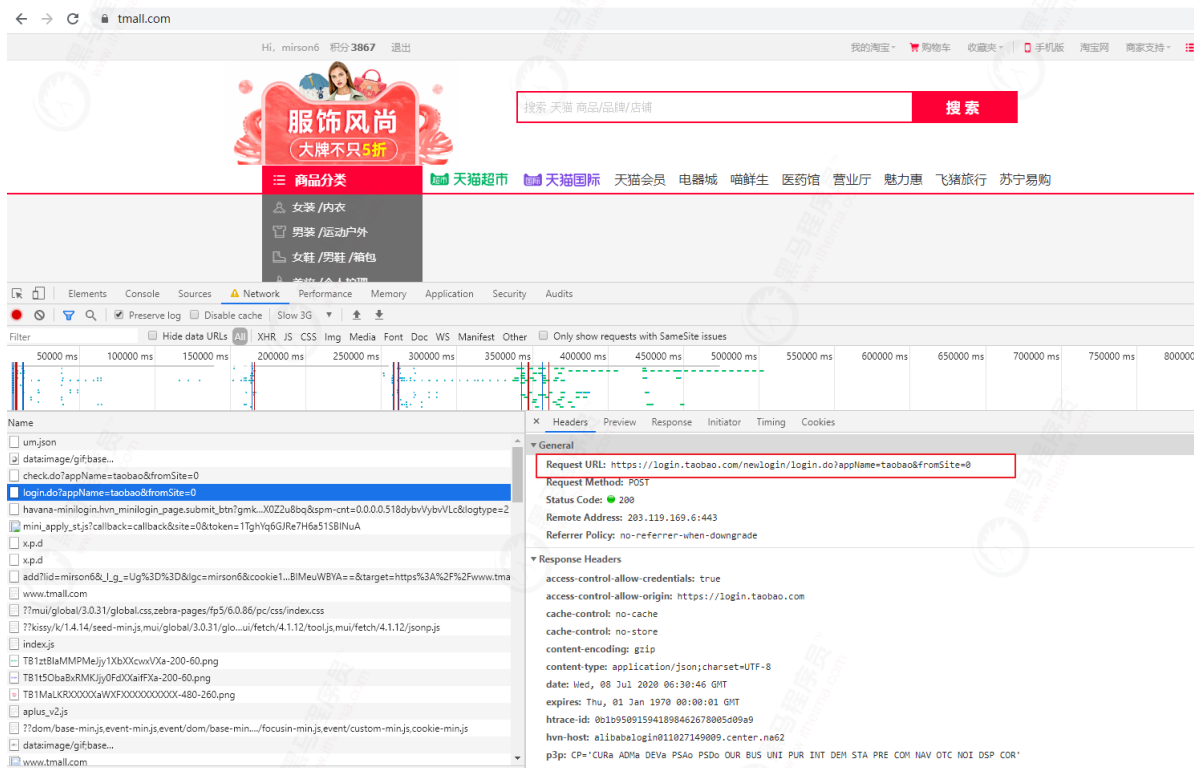


SSO登录架构设计



SSO登录实现流程解析

1. 用户进入淘宝登录页面，调用地址: <https://login.taobao.com/newlogin/login.do>
2. 调用成功之后，同步Cookie，保存Token认证信息。



2. 单点登录之整体解决方案

2.1 设计方案-Cookie

- 概述

用户登录之后, 将认证信息存储至Cookie, 当再次访问本服务或者访问其他应用服务时, 直接从Cookie中传递认证信息, 进行鉴权处理。

- 问题

1. 如何保障Cookie内用户认证信息的安全性?

第一, Cookie内不能存放用户名和密码等敏感信息, 可以生成一串Token进行替代;

第二, 通过加密方式存储Cookie信息, 并且采用https加密方式传输, 设定Cookie有效期, 在服务端设定Token的有效期, 避免攻击者伪造用户身份。

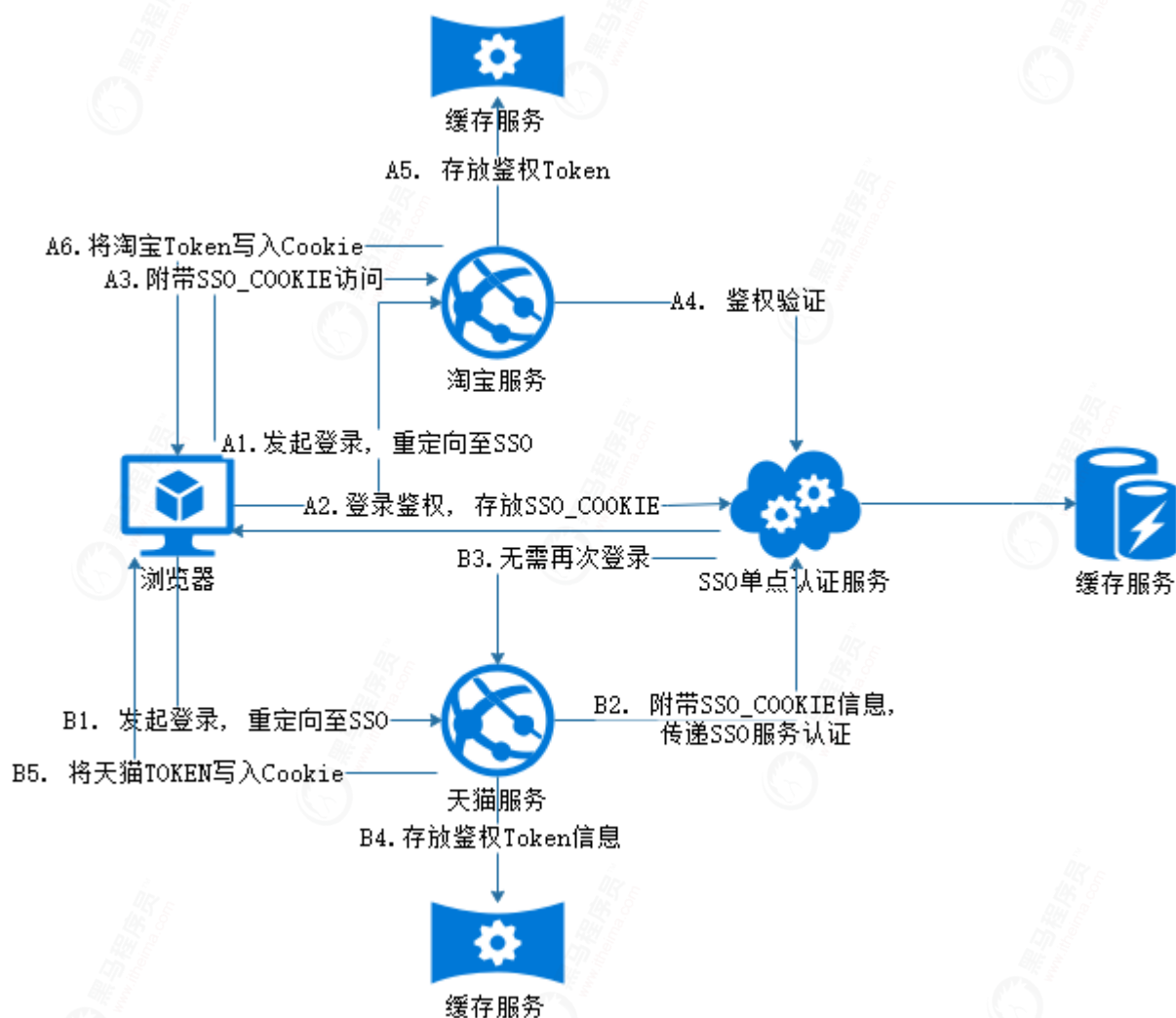
2. 如何解决跨域问题?

在实际应用中, 经常会存在各种服务需要鉴权处理, 但受浏览器同源策略限制, 无法去正常操作Cookie数据, 解决方式有两种:

第一种, 采用iframe方式解决跨域问题, 实现Cookie共享, 但要注意, 父窗口获取子窗口在跨域下可以正常获取, 子窗口后去父窗口仍会存在跨域问题, 这点在实现的时候要注意。

第二种, 采用JSONP方式实现跨域传输, 这需要在服务端设置允许跨域请求, `response.setHeader("Access-Control-Allow-Origin", "*");` 设置允许任何域名跨域访问, 服务端返回数据时, 再设置callback, 才能完成跨域请求。

- 跨域Cookie设计实现方案

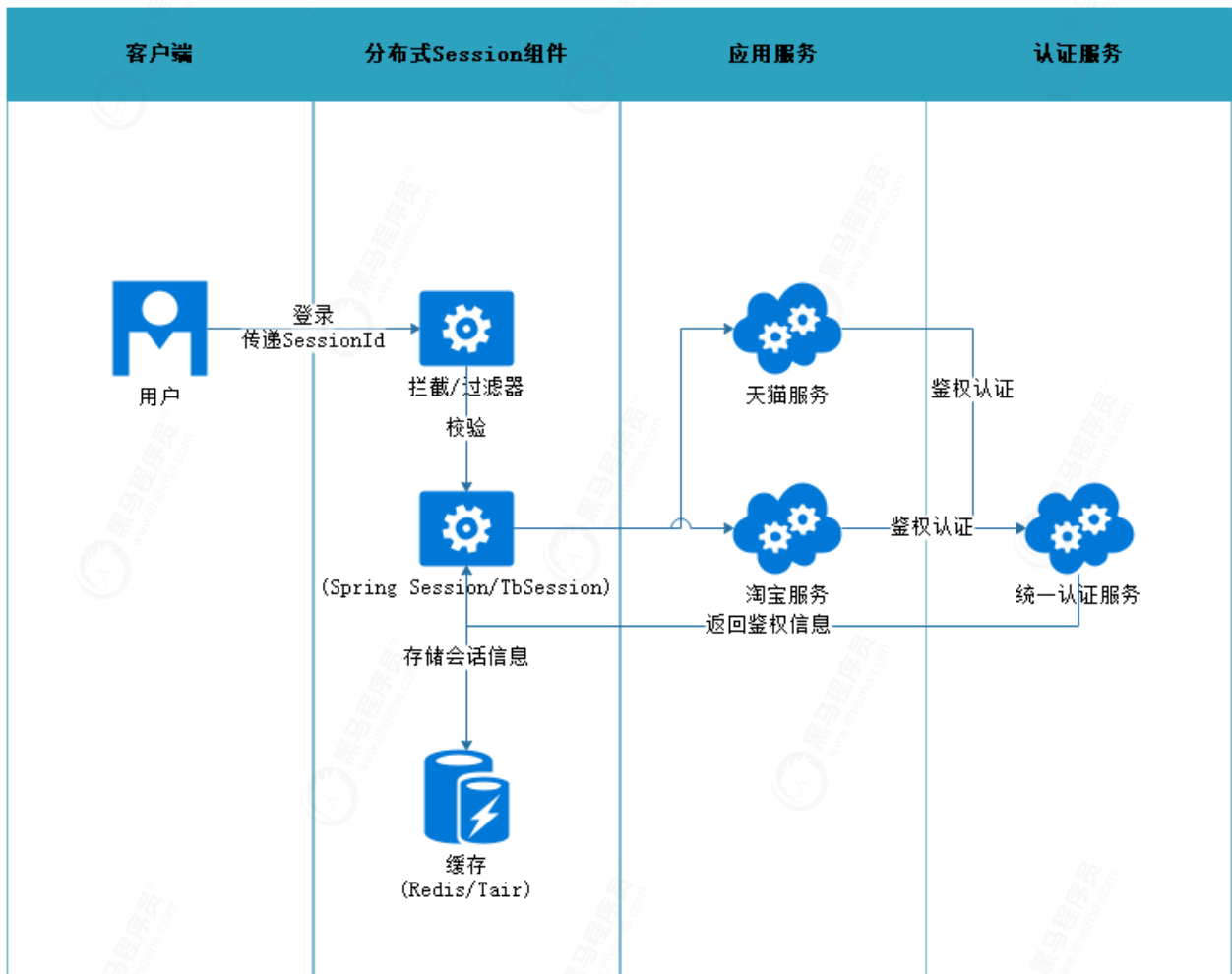


2.2 设计方案-分布式Session

- 概述

大型应用服务无论是整体拆分, 还是集群部署, 都会涉及到统一会话问题, 如何保障各服务节点都能够统一有效鉴权? 某个服务节点宕机, 重启后如何恢复登录状态? 在Cookie禁用的情况下如何实现SSO? 由此产生了分布式Session设计方案。分布式Session方案, 实质是通过自定义的Session机制来处理用户的登录鉴权信息, 实现单点登录。

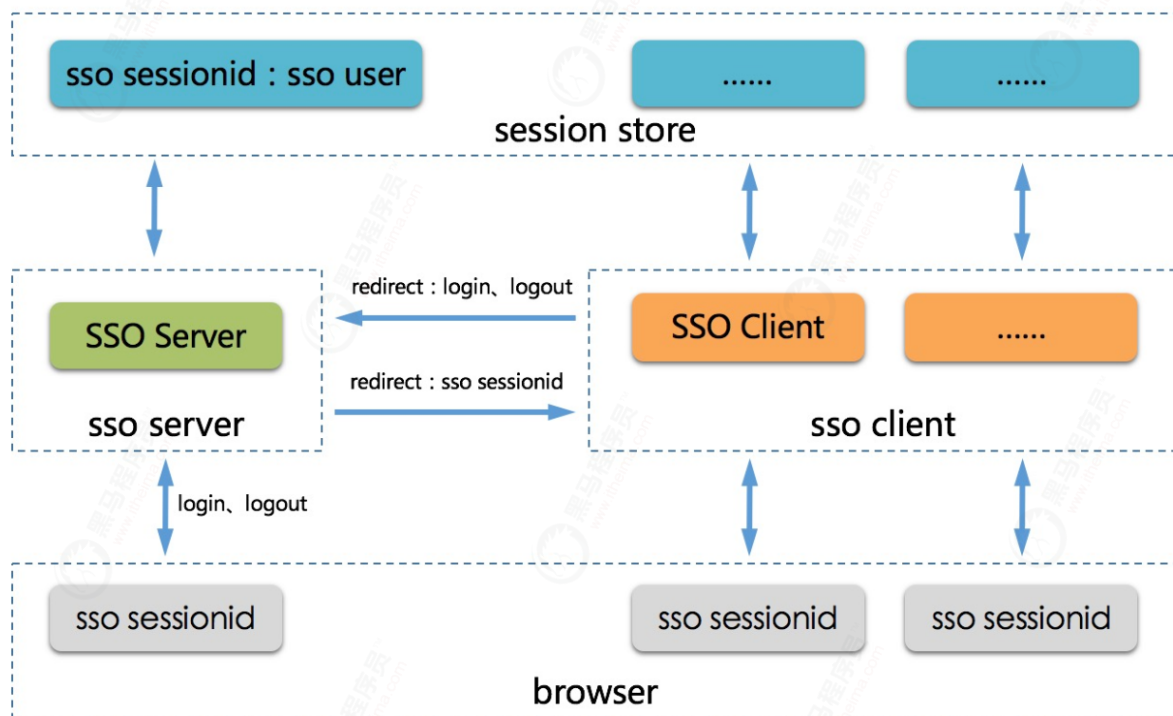
- 实现流程



- 技术框架

Spring Session : 它是目前主流的Session 管理解决方案，Spring Session 并非特定应用于HTTP， 它是一种广义的分布式统一Session，支持WebSocket和WebSession等，并且可以基于Redis、MongoDB等多种高性能缓存来实现。

XXL-SSO: 它是一个分布式单点登录框架。只需要登录一次就可以访问所有相互信任的应用系统。拥有“轻量级、分布式、跨域、Cookie+Token均支持、Web+APP均支持”等特性。现已开放源代码，开箱即用。架构图:



XXL-SSO架构图 v0.1

2.3 设计方案-客户端令牌Token

- 概述

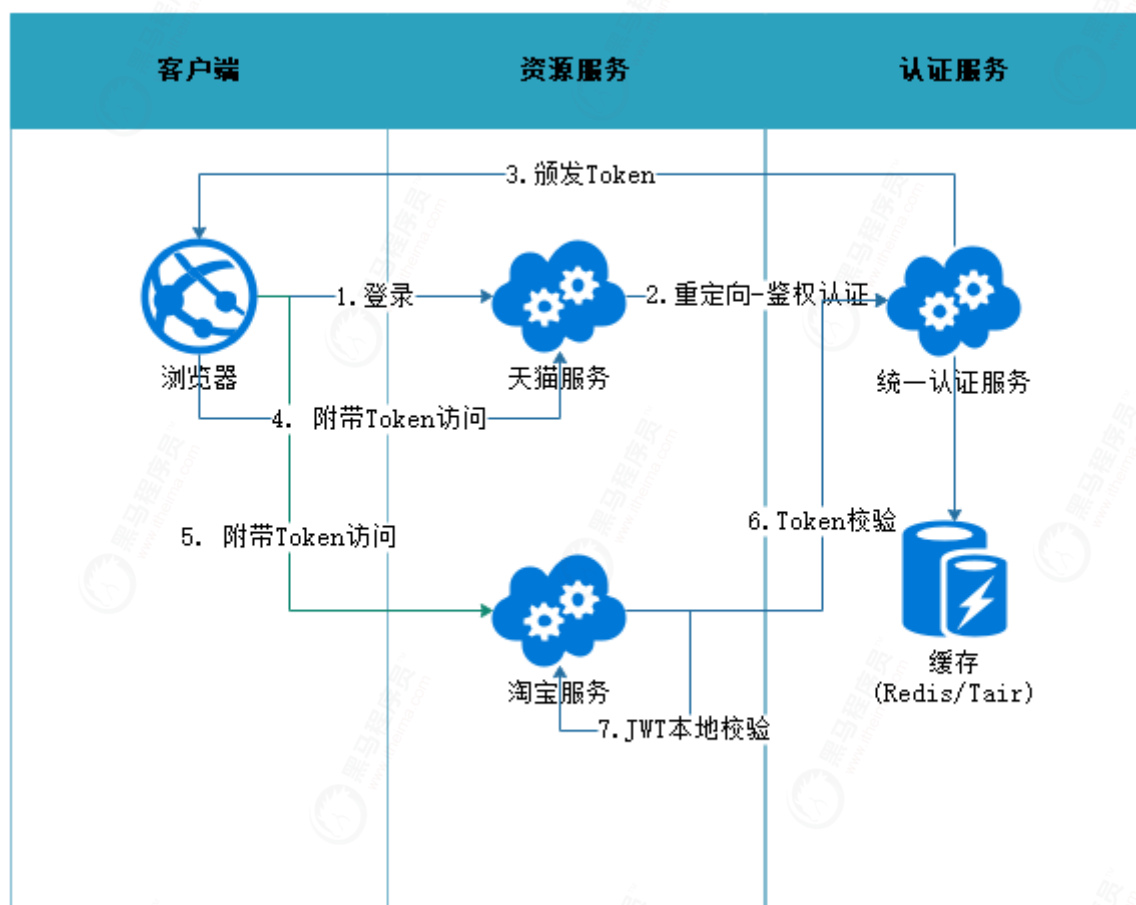
根据客户端身份信息由认证服务生成签名令牌，令牌中会包含基本的用户信息，客户端在请求资源服务时会附带令牌，资源服务根据加密协议在本地进行验证，或者发送给认证服务端进行校验。它可以解决分布式会话的安全性问题，比如会话劫持，同时不需要集中统一维护session，能够做到无状态化处理。OAuth2和JWT都是基于令牌Token实现的认证方案。

- 适用场景

JWT (JSON Web Token) 是一个开放安全的行业标准,用于多个系统之间传递安全可靠的信息。它由三部分组成，头部(Header)、载荷(payload)与签名(Signature)。Token实质是一个无意义的UUID，需要服务端做记录与认证，但JWT则赋予了用户的身份信息，可以采用自定义算法进行加密与解密，直接实现信息的传输交换。那具体适用于哪些场景？

- 可以适用于微服务应用，无论是内部服务节点的认证与授权，或是令牌与API网关结合的认证。
- 可以适用于开放式的API接口访问，比如前后分离API对接，第三方API接口对接等。

- 实现流程

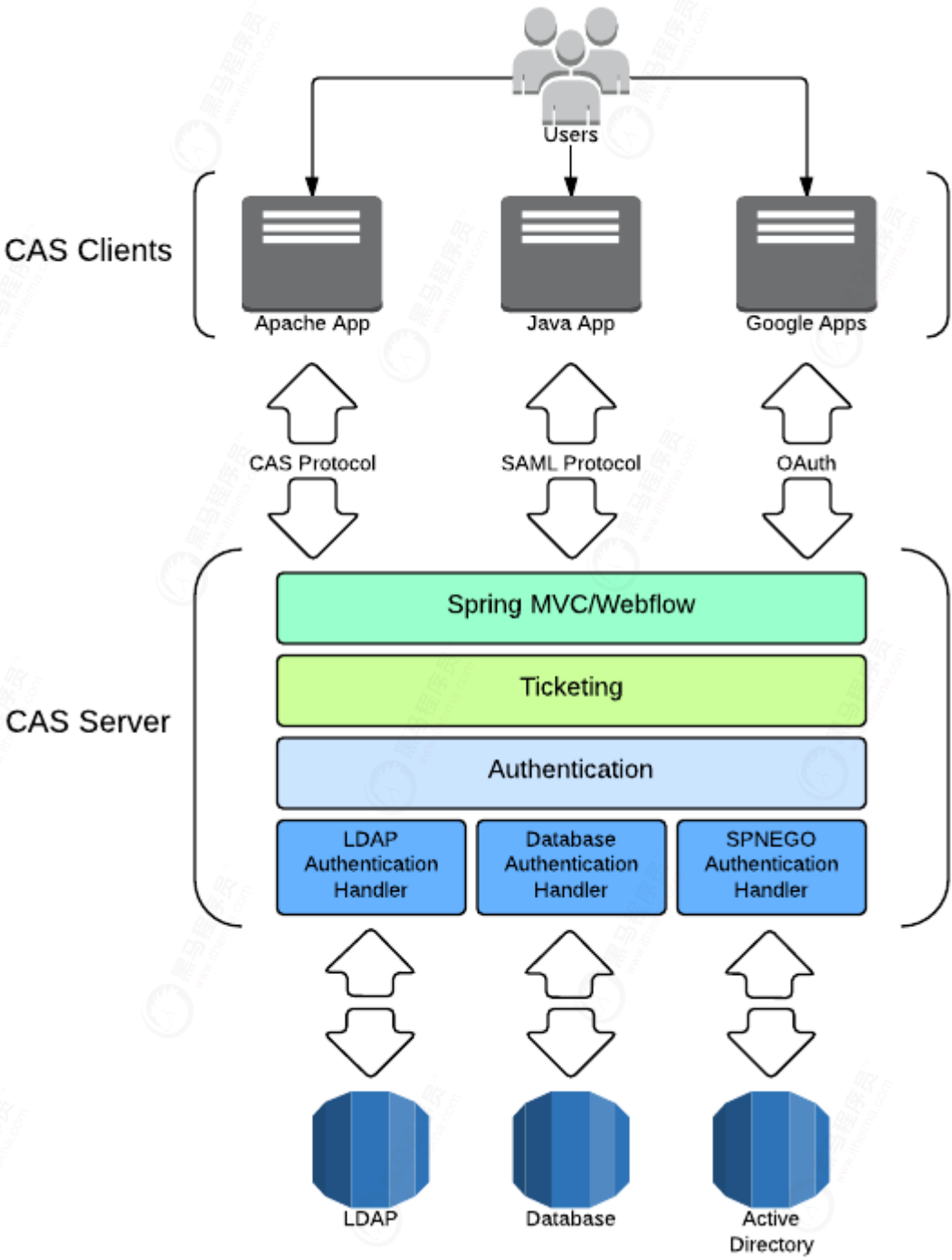


2.4 技术方案-CAS认证

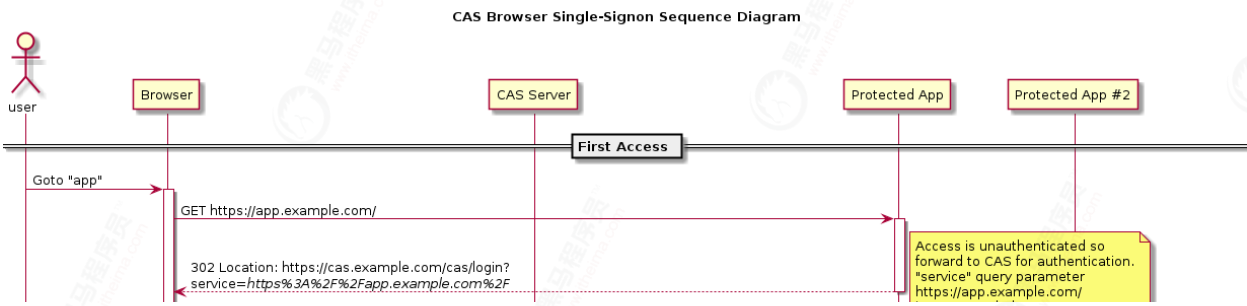
- 概述

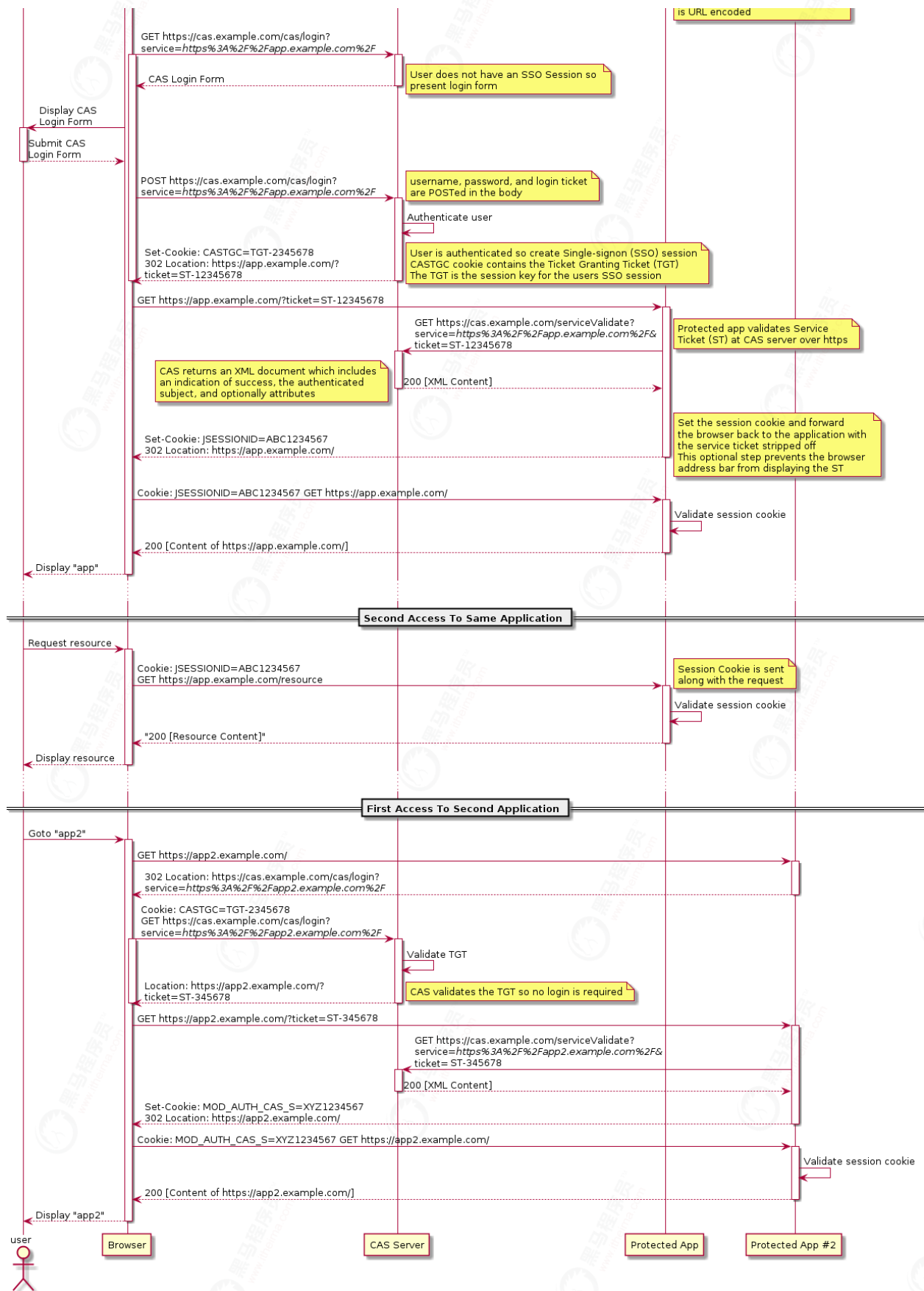
CAS（Central Authentication Service）是耶鲁大学的开源项目，宗旨是为web应用系统提供一种可靠的单点登录解决方案。CAS从安全性角度来考虑设计，用户在CAS输入用户名和密码之后通过ticket进行认证，能够有效防止密码泄露。CAS广泛使用于传统应用场景中，比如企业内部的OA，ERP等应用，不适用于微服务领

域。



设计实现流程





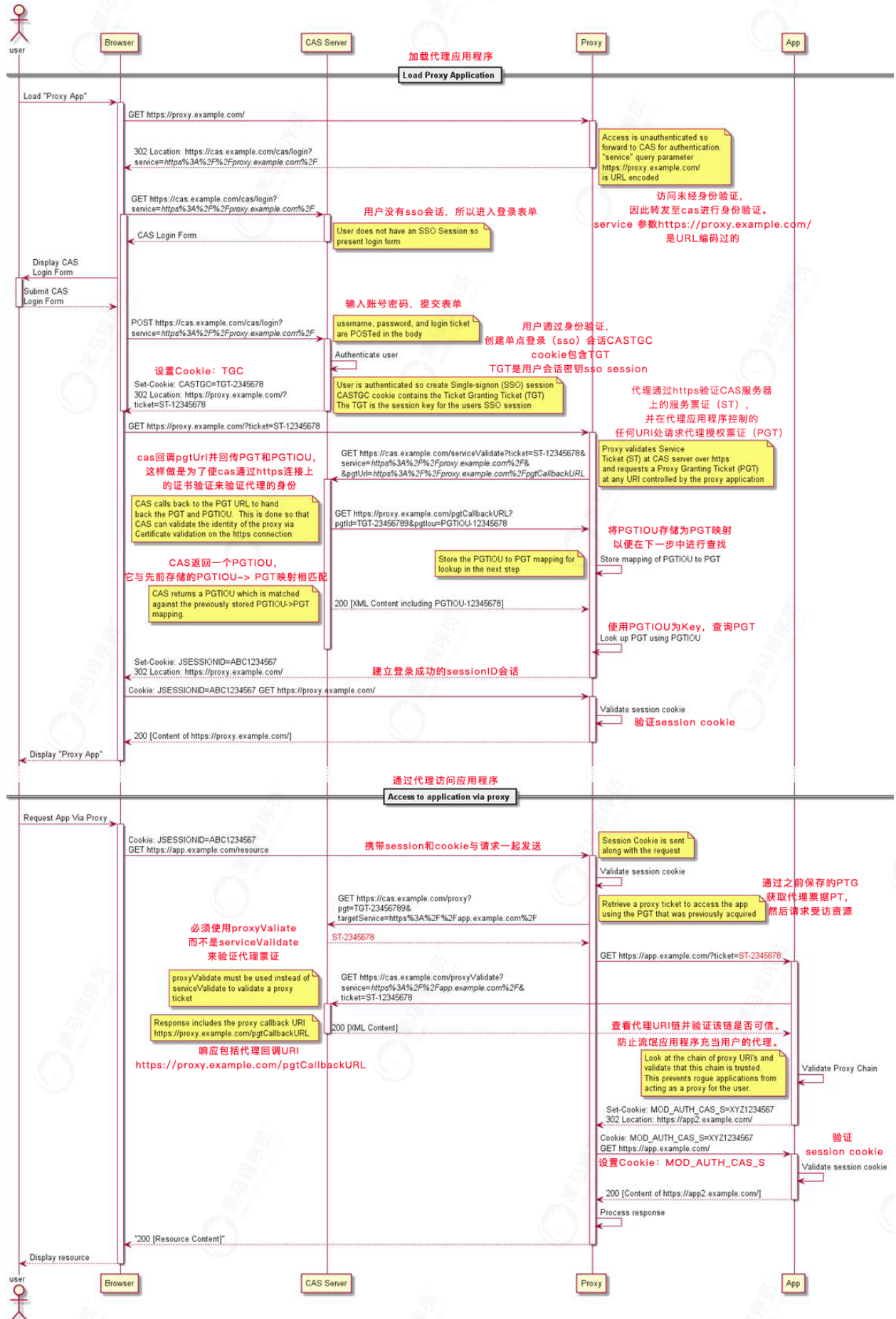
- CAS代理认证

有两个应用App1和App2，它们都是受Cas Server保护，请求它们时都需要通过Cas Server的认证。现需要在App1中以Http方式请求访问App2，显然该请求将会被App2配置的Cas的AuthenticationFilter拦截并转向Cas Server，Cas Server将引导用户进行登录认证，这样我们就不能真正的访问到App2了。针对这种应用场景，Cas也提供了对应的支持。

代理认证具体流程:

App1先通过Cas Server的认证，然后向Cas Server申请一个针对于App2的proxy ticket，之后在访问App2时把申请到的针对于App2的 proxy ticket 以参数 ticket 传递过去。App2的 AuthenticationFilter 将拦截到该请求，发现该请求携带了 ticket 参数后将放行交由后续的Ticket Validation Filter处理。Ticket Validation Filter将会传递该ticket到Cas Server进行认证，显然该ticket是由Cas Server针对于App2发行的，App2在申请校验时是可以校验通过的，这样我们就可以正常的访问App2了。

CAS Proxy Sequence Diagram



2.5 技术方案-OpenID认证

- 概述

OIDC(OpenID Connect) 是属于是OAuth 2.0协议之上的简单身份层，用API进行身份交互，允许客户端根据授权服务的认证结果确认用户的最终身份，它支持包括Web、移动、JavaScript在内的所有客户端类型。它与OAuth的主要区别是在于, OpenID 只用于身份认证，例如允许一个账户登录多个网站；而OAuth可以用于授权，允许授权的客户端访问指定的资源服务。

- 应用场景

如果有独立账号体系，需要为外部提供统一认证服务， 可以采用OIDC，OIDC目前有很多企业在使用，比如Google的账号认证体系，Microsoft的账号体系也采用了OIDC。

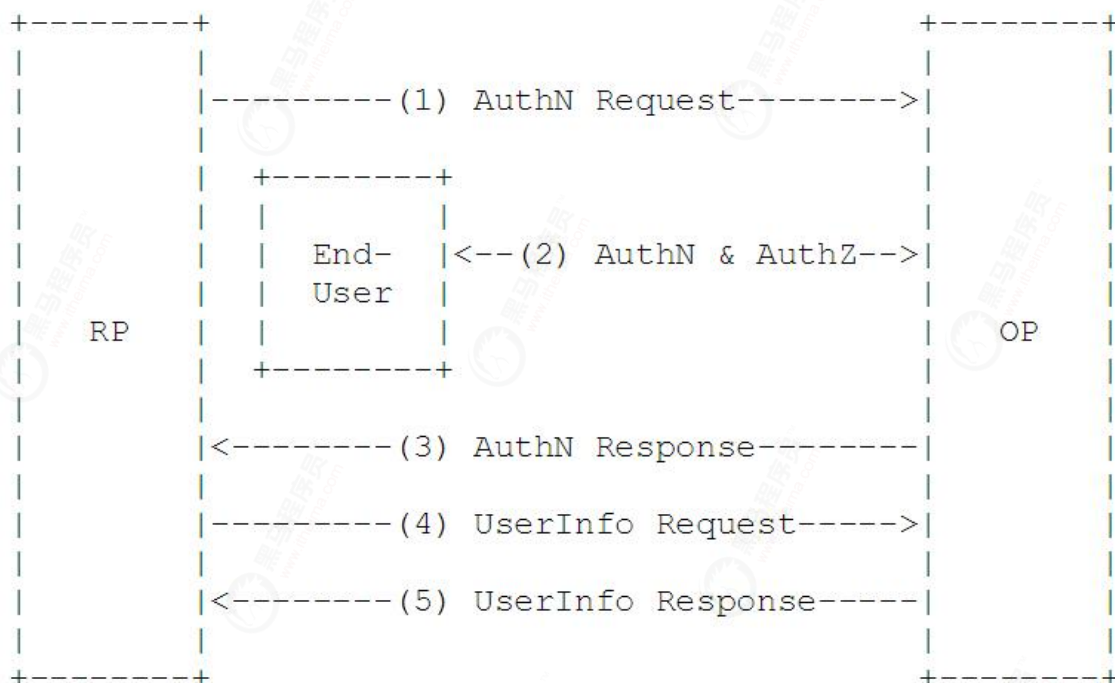
- 如何工作

OAuth2提供了Access Token来解决授权第三方客户端访问受保护资源的问题；OIDC在这个基础上提供了ID Token来解决第三方客户端标识用户身份认证的问题。OIDC的核心在于在OAuth2的授权流程中，一并提供用户的身份认证信息（ID Token）给到第三方客户端，ID Token使用JWT格式来包装，得益于JWT（JSON Web Token）的自包含性，紧凑性以及防篡改机制，使得ID Token可以安全的传递给第三方客户端程序并且容易被验证。此外还提供了UserInfo的接口，用户获取用户的更完整的信息。

- 工作流程

术语解析:

1. EU（End User）：代表终端用户。
2. RP（Relying Party）：指OAuth2中受信任的客户端。
3. OP（OpenID Provider）：有能力提供EU认证的服务（比如OAuth2中的授权服务），为RP提供EU的身份认证信息。
4. ID Token: JWT格式的数据，包含EU身份认证的信息。
5. UserInfo Endpoint: 用户信息接口（受OAuth2保护），当RP使用Access Token访问时，返回授权用户的信息，此接口必须使用HTTPS。



- 工作模式

1. 默认模式/简化模式（Implicit Flow）：如果是Web应用服务，其所有的代码都有可能被加载到浏览器暴露出来，无法保证终端client_secret的安全性，则采用默认模式。
2. 授权码模式（Authentication Flow）：如果是传统的客户端应用，后端服务和用户信息是隔离的，能保证client_secret的不被泄露，就可以使用授权码模式流程。
3. 混合模式（Hybrid Flow）：实质上是以上两种模式的融合，混合模式下ID Token通过浏览器的前端通道传递，而Access Token和Refresh Token通过后端获取，混合使用，可以弥补两种模式的缺点，一般推荐使用混合模式。

2.6 技术方案-SAML2.0认证

- 什么是SAML

SAML 全称是 Security Assertion Markup Language。SAML是支持身份认证的协议，它可以通过支持XACML协议进行权限控制。SAML是基于XML实现的协议，较OAUTH来说较复杂些，但功能也十分强大，支持认证，权限控制和用户属性识别等。目前在云服务的接入使用比较广泛，作为重点内容，在下面的章节做详细讲解。

2.7 技术方案-OAuth2认证

- 什么是OAuth

OAuth 2.0 是一个行业的标准授权协议，它的最终目的是为第三方应用颁发一个有时效性的令牌 token，使得第三方应用能够通过该令牌获取相关的资源。它的主要作用可以实现登录认证与授权，常见的场景：比如第三方登录，当你要登录某个论坛，但没有账号，通过QQ登录的过程就是采用 OAuth 2.0 协议，通过OAuth2的授权，可以获取QQ头像等资源信息。OAuth2是目前应用最为广泛的认证授权协议，这是重点内容，在下面的章节做详细深入讲解。

3. 单点登录之技术方案深入详解

3.1 基于SAML实现的统一认证

3.1.1 概述

SAML 2.0 用来在安全域中交换身份验证（Authentication）数据和授权（Authorization）数据。SAML 2.0基于XML协议，使用包含断言（Assertions）的安全令牌在SAML授权方（即身份提供者IdP）和SAML消费方（即服务提供者SP）之间传递委托人（终端用户）的信息。

SAML 2.0 可以实现基于网络跨域的单点登录(SSO)，以便于减少向一个用户分发多个身份验证令牌的管理开销。

3.1.2 什么是断言（Assertions）

断言是一个包含了由SAML授权方提供的0到多个声明（statement）的信息包。SAML断言通常围绕一个主题生成。该主题使用声明。SAML 2.0规范定义了三种断言声明，详细信息如下：

- 身份验证（Authentication）：该断言的主题是在某个时间通过某种方式被认证。
- 属性（Attribute）：该言的主题和某种属性相关联。
- 授权决策（Authorization Decision）：该断言的主题被允许或者被禁止访问某个资源。

断言举例：

```
<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ID="b07b804c-7c29-ea16-7300-4f3d6f7928ac"
  Version="2.0"
  IssueInstant="2004-12-05T09:22:05Z">
  <saml:Issuer>https://idp.example.org/SAML2</saml:Issuer>
  <ds:Signature
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">...</ds:Signature>
  <saml:Subject>
    <saml:NameID
      Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient">
      3f7b3dcf-1674-4ecd-92c8-1544f346baf8
    </saml:NameID>
    <saml:SubjectConfirmation
      Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <saml:SubjectConfirmationData
        InResponseTo="aaf23196-1773-2113-474a-fe114412ab72"
        Recipient="https://sp.example.com/SAML2/SSO/POST"
        NotOnOrAfter="2004-12-05T09:27:05Z"/>
      </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions
      NotBefore="2004-12-05T09:17:05Z"
      NotOnOrAfter="2004-12-05T09:27:05Z">
      <saml:AudienceRestriction>
        <saml:Audience>https://sp.example.com/SAML2</saml:Audience>
      </saml:AudienceRestriction>
    </saml:Conditions>
    <saml:AuthnStatement
      AuthnInstant="2004-12-05T09:22:00Z"
      SessionIndex="b07b804c-7c29-ea16-7300-4f3d6f7928ac">
      <saml:AuthnContext>
        <saml:AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes>PasswordProtectedTransport
        </saml:AuthnContextClassRef>
      </saml:AuthnContext>
    </saml:AuthnStatement>
    <saml:AttributeStatement>
      <saml:Attribute
        xmlns:x500="urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
        x500:Encoding="LDAP"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
        Name="urn:oid:1.3.6.1.4.1.5923.1.1.1.1"
        FriendlyName="eduPersonAffiliation">
        <saml:AttributeValue
          xsi:type="xs:string">member</saml:AttributeValue>
        <saml:AttributeValue
          xsi:type="xs:string">staff</saml:AttributeValue>
        </saml:Attribute>
      </saml:AttributeStatement>
    </saml:Assertion>
```

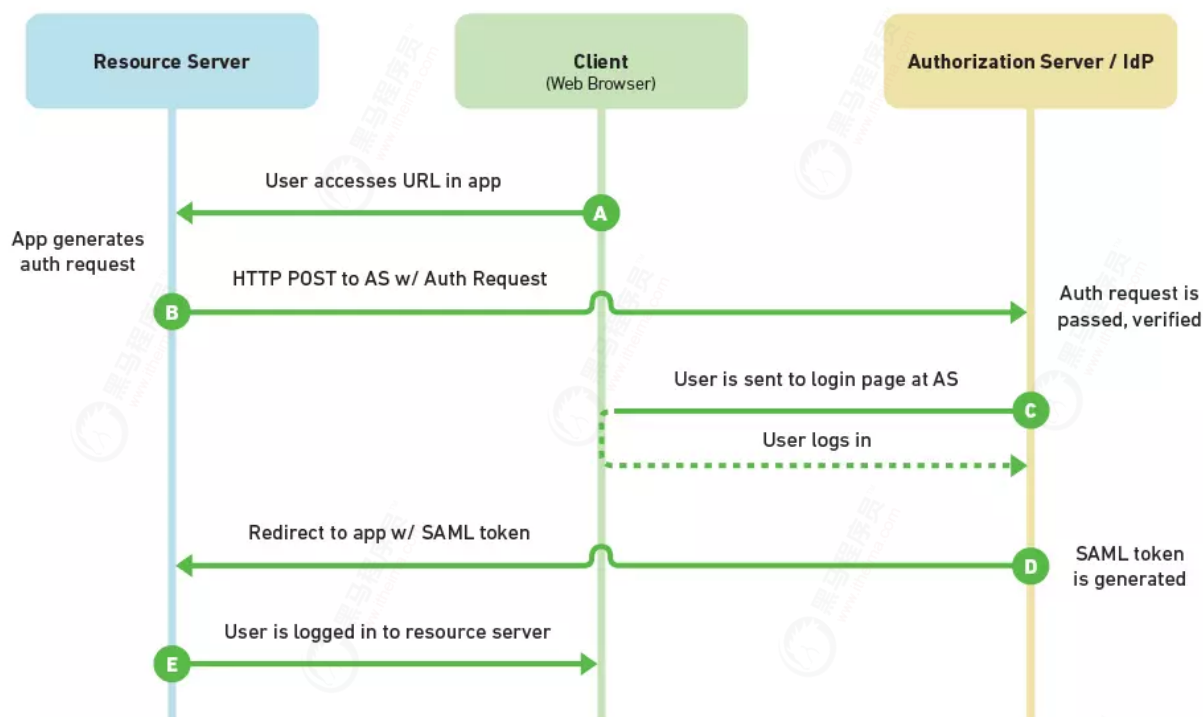
它主要是用来实现Web浏览器的单点登录。该断言包括一个身份验证断言 `saml:AuthnStatement` 和一个属性断言 `saml:AttributeStatement`，SP将使用该属性断言实现访问控制。

3.1.3 工作流程

SAML 认证流程一般都会牵涉到两方：服务提供方（SP）和身份提供方（IdP），典型的 SP 有阿里云、腾讯云以及很多很多的 SaaS 服务；IdP 其实就是我们企业自己，因为用户目录在我们这里。

访问 SP 服务的时候，SP 会向 IdP 发送一个 SAML Request（具体是什么我们暂时不关心），请求 IdP 判断用户身份。IdP 收到 SAML Request 后，可以通过某种手段对用户身份进行认证，如果已登录，可以直接返回用户身份信息给 SP；如果未登录，可以弹出一个登录框，用户登录之后再用户身份返回给 SP。SP 收到用户信息之后，再在自己的数据库里面找出对应的用户，然后以这个用户的身份访问 SP 服务。

SAML 2.0 Flow



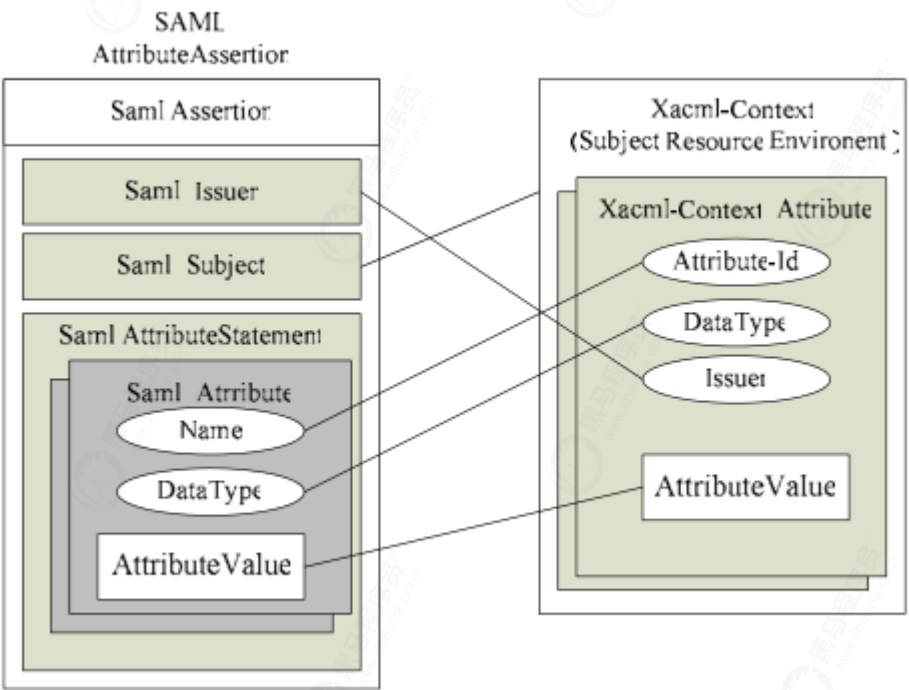
1. 用户通过浏览器访问网站（SP），网站提供服务但是并不负责用户认证。
2. SP 向 IDP 发送了一个 SAML 认证请求，同时 SP 将用户浏览器重定向到 IDP。
3. IDP 在验证完来自 SP 的合法请求，在浏览器中呈现登陆表单让用户填写用户名与密码信息，进行登陆。
4. 用户登陆成功，IDP 会生成一个包含用户信息的 SAML token（SAML token 又称为 SAML Assertion，本质上是 XML 节点）。IDP 向 SP 返回 token，并且将用户重定向到 SP。
5. SP 对拿到的 token 进行验证，并从中解析出用户信息，例如用户是谁以及用户的权限有哪些。此时可以根据这些授权信息允许用户访问我们网站的内容。

3.1.4 授权机制

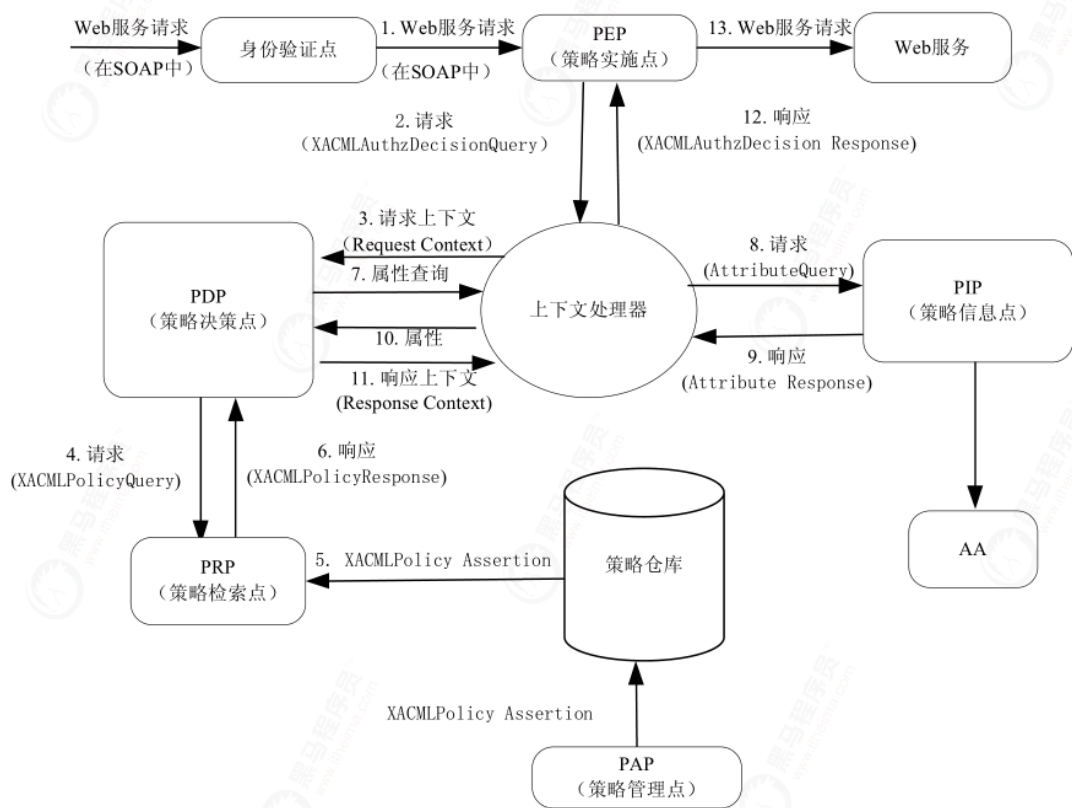
SAML 只是认证协议，自身并不提供授权功能，可以通过 XACML 实现授权。

XACML 是可扩展访问控制标记语言，以 XML 的形式描述策略语言和授权决策请求/响应，提供管理授权决策的语法。

SAML 和 XACML 结合实现权限访问控制，映射关系：



SAML 和 XACML 结合控制应用模型：



该模型是一个完整的访问控制体系结构，包含身份验证和授权两部分。身份验证可以接受来自其它系统的各种安全令牌，包括 SAML 断言，对请求主体进行验证并产生 SAML 身份验证断言。只要合作的第三方服务联合信任，就可以实现服务的安全交互以及用户的单点登录。

模型的授权基于 PMI 统一授权管理体系，授权系统向 AA（属性权威机构）请求关于 Web 服务请求主体的属性信息，AA 实现 SAML 接口，返回 SAML 属性断言。

模型使用统一的策略语言 XACML，由 SAML 为其提供底层传输机制，适用于各种类型的访问控制系统。策略可以被不同的应用使用，使策略的管理更加容易。

3.1.5 应用场景

目前SAML广泛应用于云服务的认证，比如阿里云、AWS和腾讯云等，在云服务上面维护统一的用户信息进行身份认证。SAML认证一般分为两部分，用户池与角色身份池。

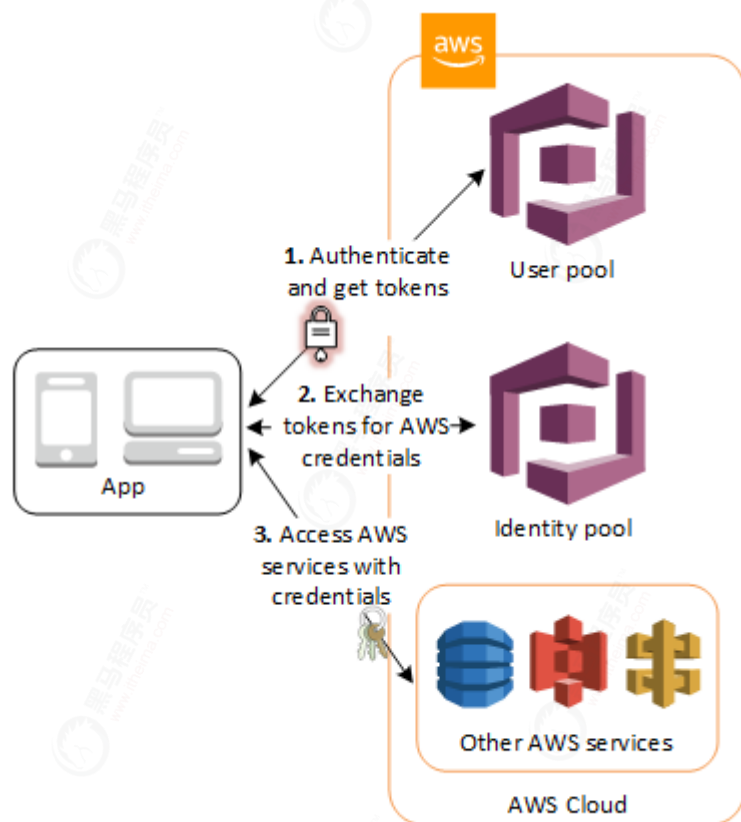
用户池可以让应用程序接入，也可以通过第三方身份提供商 (IdP)，对用户身份进行认证。

角色身份池可以通过凭证来控制访问云服务资源，比如阿里云推送服务，Amazon S3 和 DynamoDB等。

以AWS的Amazon Cognito为例，简单了解下它的应用：

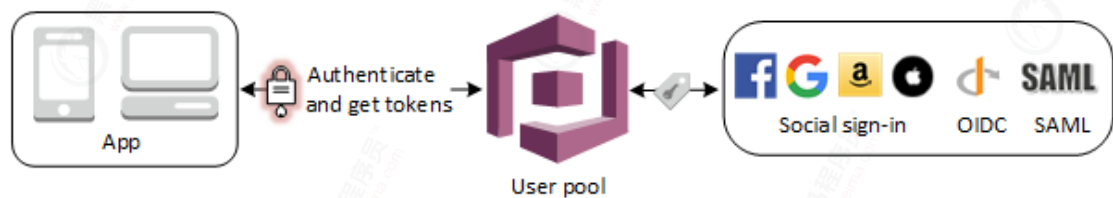
通过SAML协议验证用户身份，然后授予用户访问其他 AWS 服务的权限。

1. 在第一步中，您的应用程序用户通过用户池登录，并在成功进行身份验证后收到用户池令牌。
2. 接下来，您的应用程序通过用户池令牌交换 AWS 凭证。
3. 最后，您的应用程序用户可以使用这些 AWS 凭证来访问其他 AWS 服务（如 Amazon S3 或 DynamoDB）。



3.1.6 AWS云服务接入方案

- 用户池进行身份验证

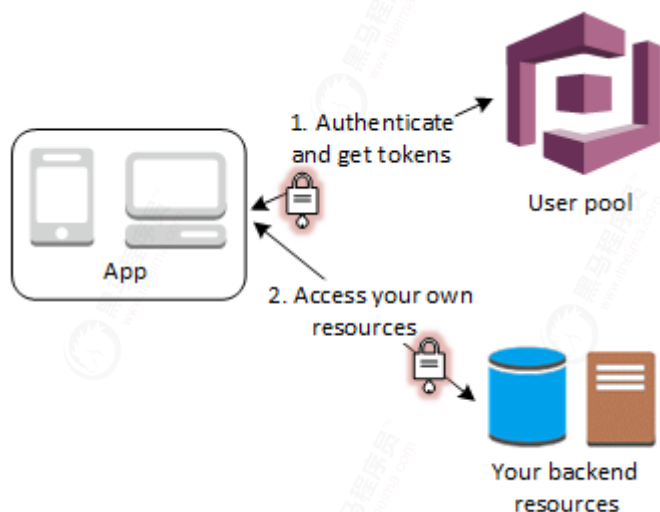


用户使用用户池进行身份验证。应用程序用户可以通过用户池直接登录，也可以通过第三方身份提供商 (IdP) 联合。用户池管理从通过 Facebook、Google、Amazon 和 Apple 进行的社交登录返回的以及从 OpenID Connect (OIDC) 和 SAML IdP 返回的令牌的处理开销。

成功进行身份验证后，Web 或移动应用程序将收到来自 Amazon Cognito 的用户池令牌。可以使用这些令牌检索允许的应用程序访问其他 AWS 服务的 AWS 凭证，也可以选择使用它们来控制对您的服务器端资源或 Amazon API Gateway 的访问。

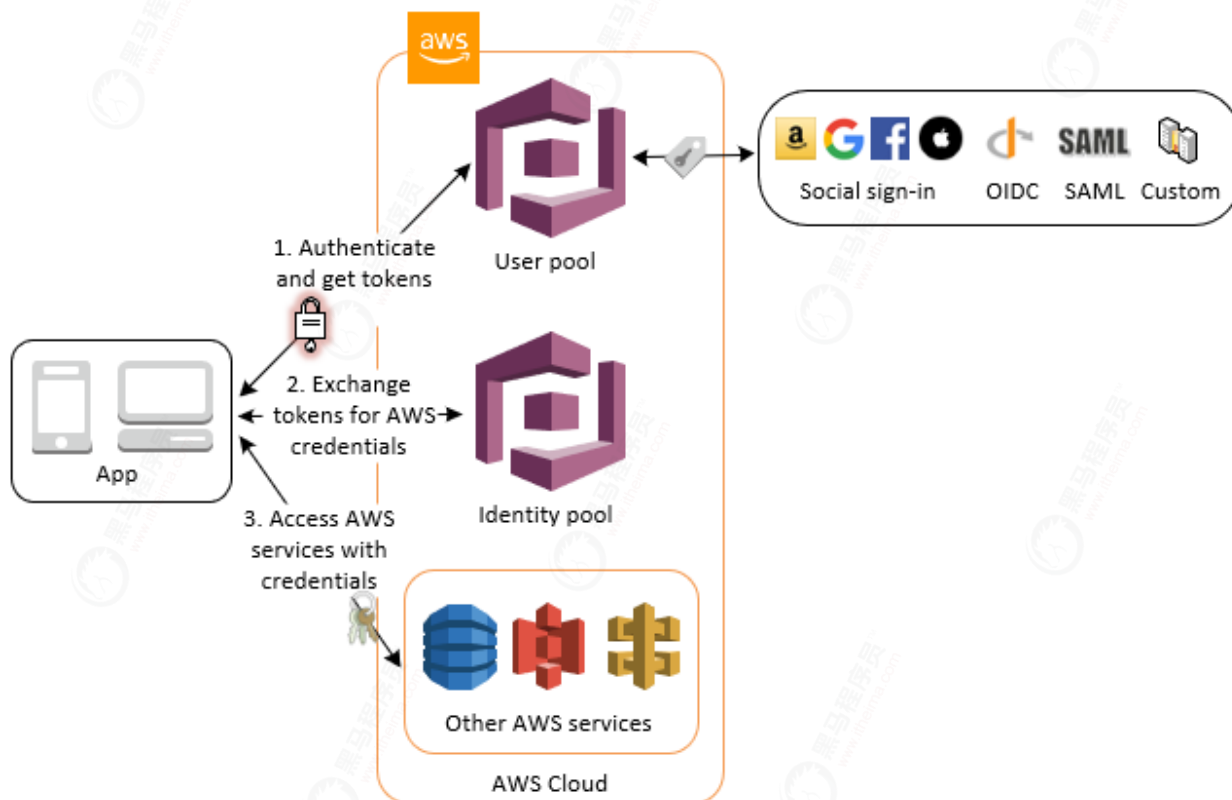
- 用户池访问服务器端资源

用户池登录后，Web 或移动应用程序将收到用户池令牌。可以使用这些令牌控制对服务器端资源的访问。可以创建用户池组来管理权限以及表示不同类型的用户。



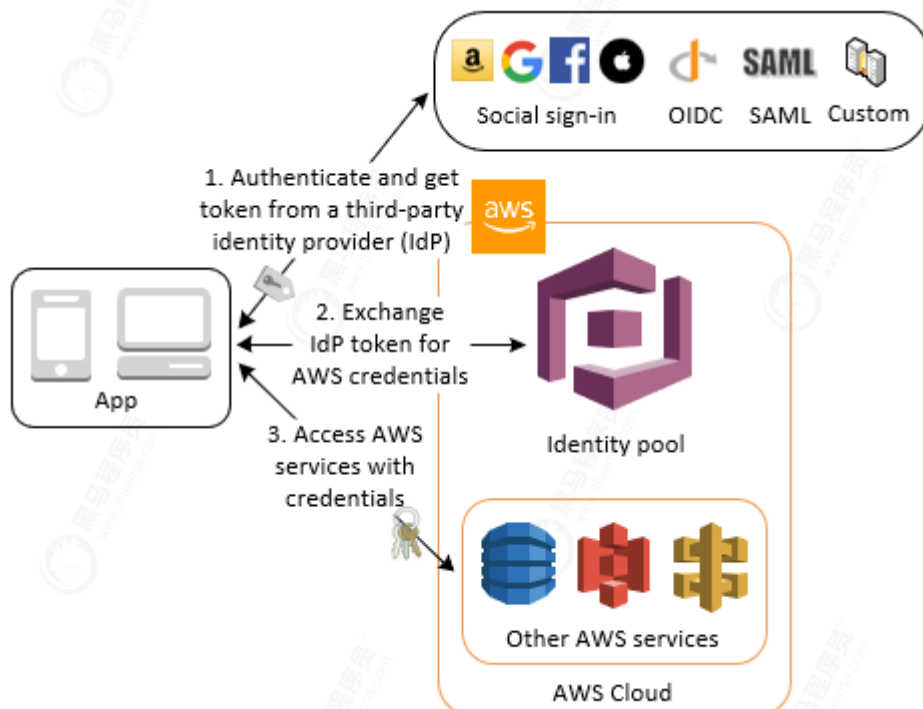
- 用户池和身份池访问云服务

用户池登录认证成功之后，获取返回的令牌，再通过令牌换取身份池的信息，拿去身份池信息就可以访问其他的云服务资源。



- 支持第三方进行身份验证并使用身份池访问云服务

身份池需来自第三方身份提供商，进行身份验证之后，返回用户的 IdP 令牌。再通过令牌交换获取云服务的身份池信息，身份池将授予可用来访问其他云服务的临时凭证。



更多资料参照官方文档:

[Amazon Cognito 教程](#)

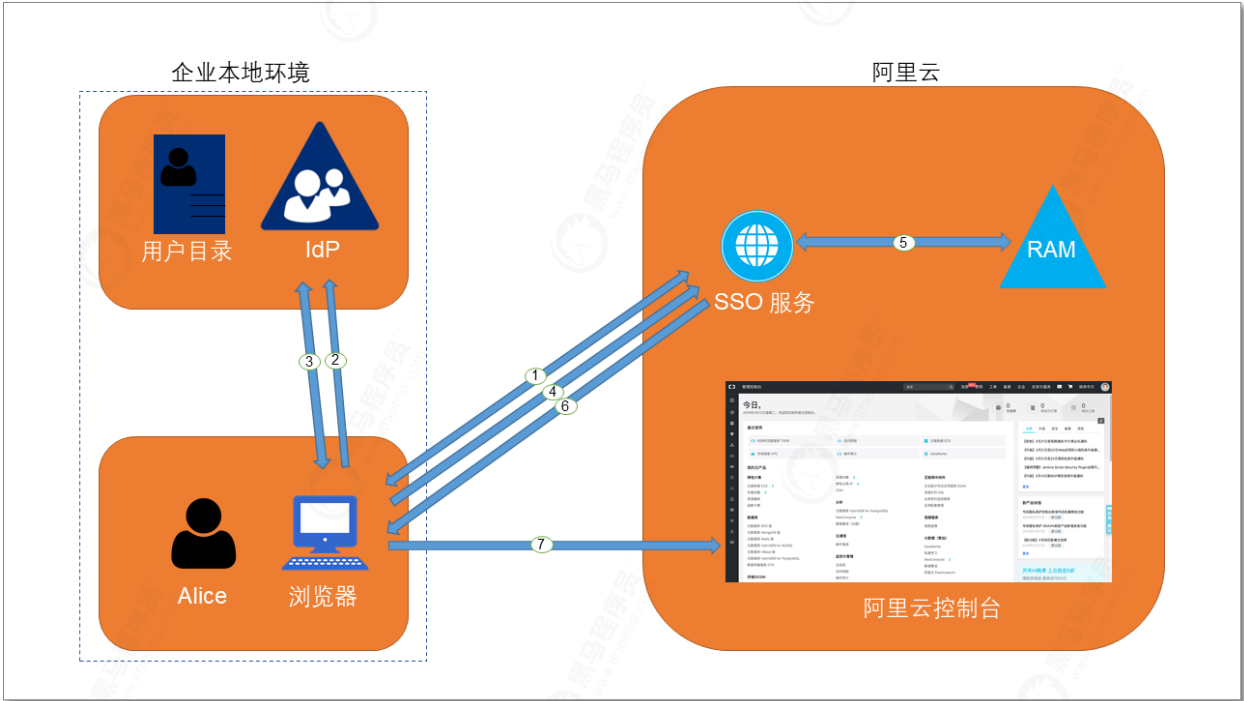
3.1.7 阿里云接入方案

阿里云支持基于SAML 2.0的SSO（Single Sign On，单点登录），也称为身份联合登录。

阿里云提供以下两种基于SAML 2.0协议的SSO方式：

用户SSO： 阿里云通过IdP颁发的SAML断言确定企业用户与阿里云RAM用户的对应关系。企业用户登录后，使用该RAM用户访问阿里云。**角色SSO：** 阿里云通过IdP颁发的SAML断言确定企业用户在阿里云上可以使用的RAM角色。企业用户登录后，使用SAML断言中指定的RAM角色访问阿里云。请参见进行角色SSO。

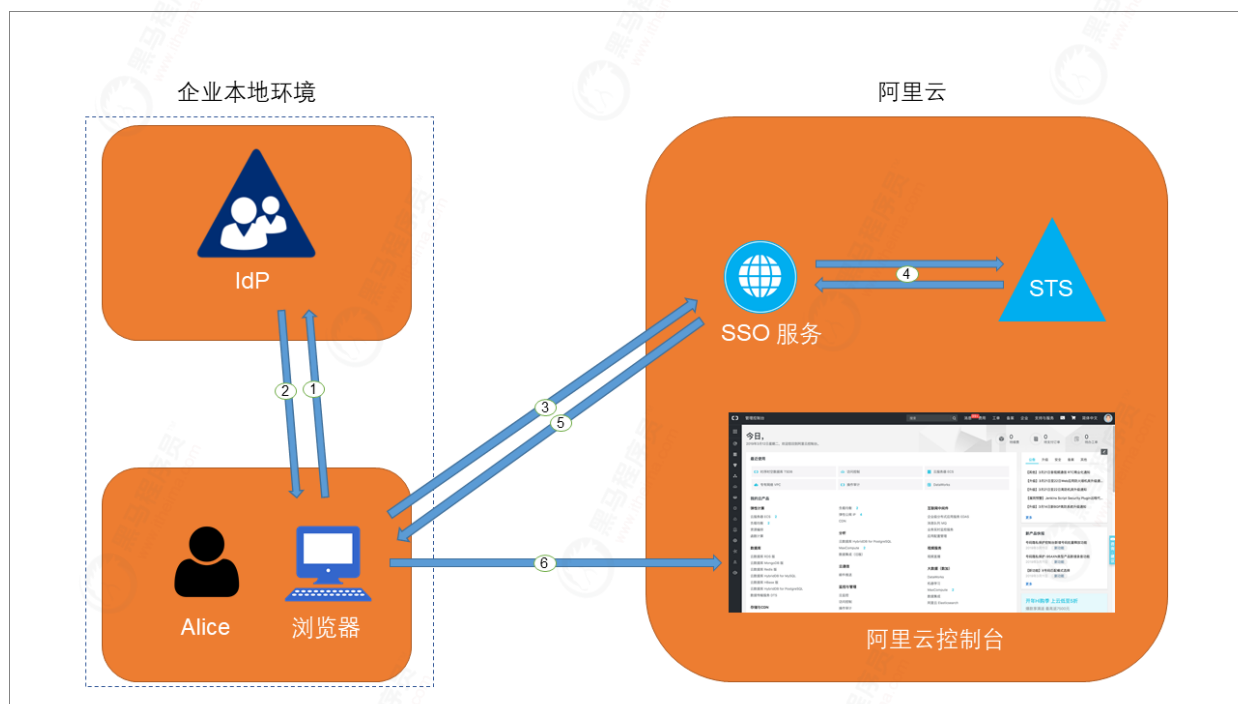
• 用户SSO



当管理员在完成用户SSO的相关配置后，可以通过以下流程来实现用户SSO。

1. Alice使用浏览器登录阿里云，阿里云将SAML认证请求返回给浏览器。
2. 浏览器向IdP转发SAML认证请求。
3. IdP提示Alice登录，并在Alice登录成功后生成SAML响应返回给浏览器。
4. 浏览器将SAML响应转发给SSO服务。
5. SSO服务通过SAML互信配置，验证SAML响应的数字签名来判断SAML断言的真伪，并通过SAML断言的NameID元素值，匹配到对应阿里云账号中的RAM用户身份。
6. SSO服务向浏览器返回控制台的URL。
7. 浏览器重定向到阿里云控制台。

• 角色SSO



1. 企业员工Alice可登录到阿里云，使用浏览器在IdP的登录页面中选择阿里云作为目标服务。
2. IdP生成一个SAML响应并返回给浏览器。
3. 浏览器重定向到SSO服务页面，并转发SAML响应给SSO服务。
4. SSO服务使用SAML响应向阿里云STS服务请求临时安全凭证，并生成一个可以使用临时安全凭证登录阿里云控制台的URL。
5. SSO服务将URL返回给浏览器。
6. 浏览器重定向到该URL，以指定角色身份登录到阿里云控制台。

更多资料参照官方文档：

[阿里云SSO](#)

3.2 基于OAuth实现的统一认证

3.2.1 概述

OAuth2 实质是为第三方应用颁发一个具有时效性的Token令牌，使其他服务或第三方应用能够通过令牌获取相关资源。常见的场景：比如进入某个网站没有账号信息，但可以通过QQ、微信、支付宝等账号进行登陆，在这个登陆过程中采用的就是Oauth2协议；OAUTH2不仅支持认证，还具备授权功能，比如通过QQ登录获取用户头像，基本资料等。

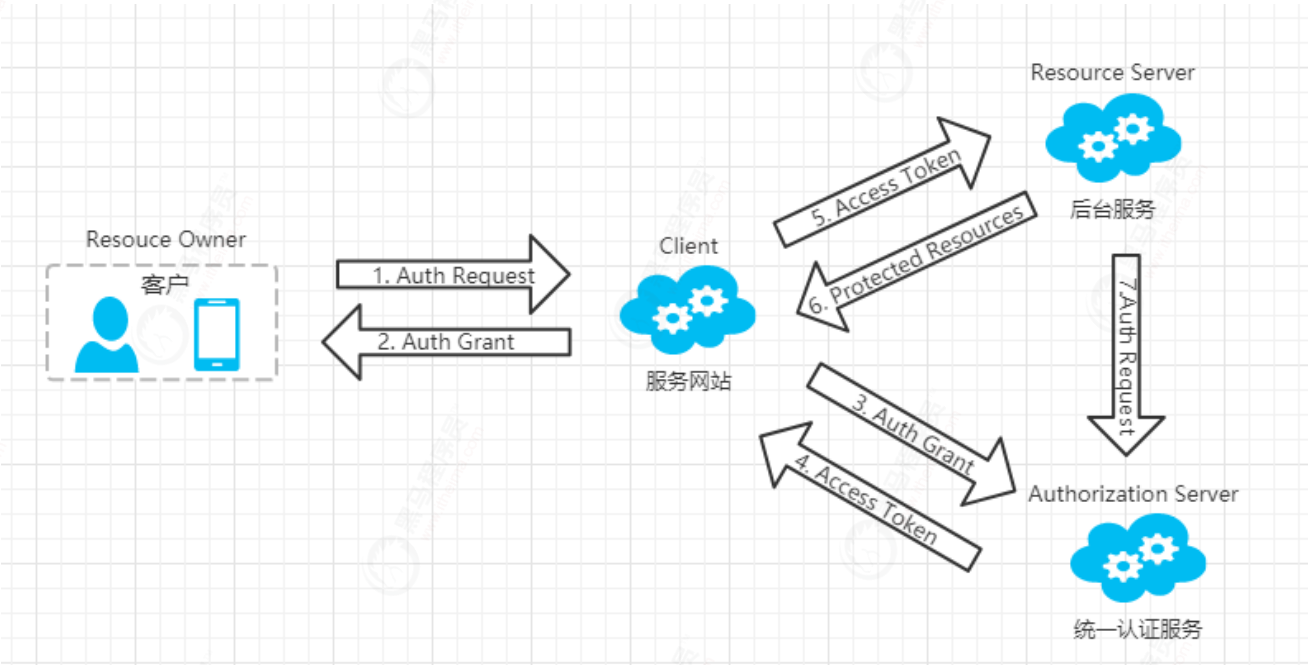
3.2.2 OAuth2角色

- resource owner：资源所有者，具备访问该资源的实体，如果是某个人，被称为end-user。
- resources server: 资源服务器，受保护的资源服务器，具备提供资源能力，如订单服务，商品服务等。
- client: 客户端，这并不是指用户，而是对资源服务器发起请求的应用程序，比如前后分离项目，前端服务访问管理接口，访问后台业务功能接口。
- authorization server: 授权服务器，能够给客户端颁发令牌，这个就是我们上面所讲的统一认证授权服务器。
- user-agent: 用户代理，作为资源所有者与客户端沟通的工具，比如APP，浏览器等。

3.2.3 OAuth2 协议流程

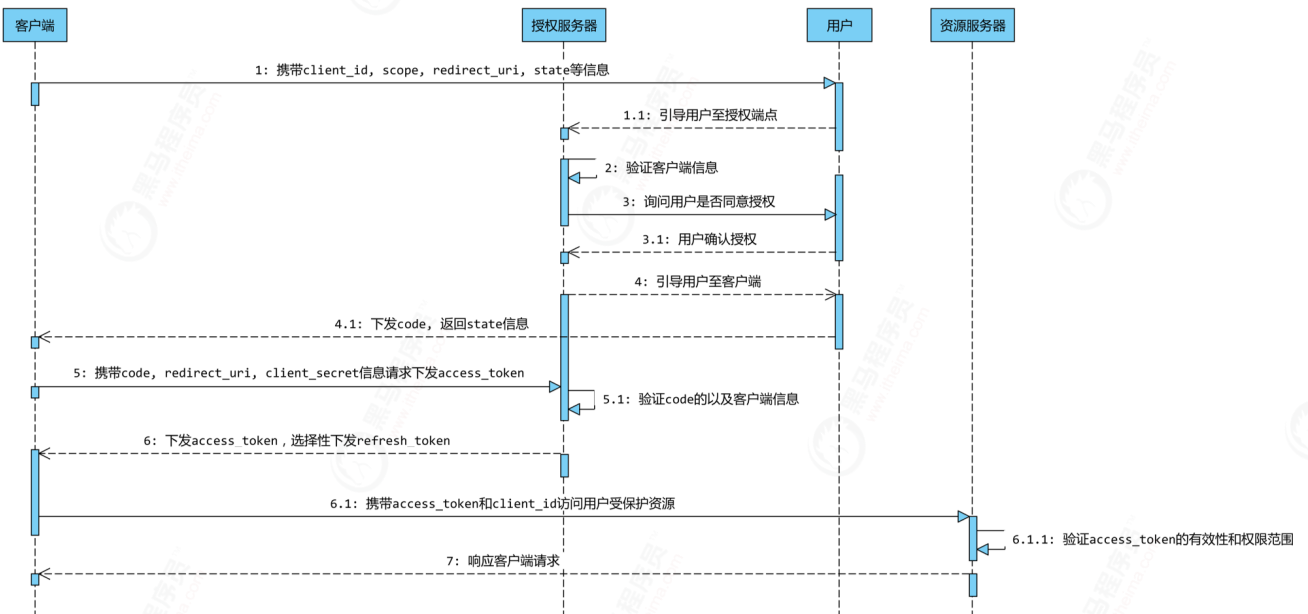
OAuth2包含四种授权模式：

- 1. 授权码模式；
- 2. 隐式/简化授权模式；
- 3. 密码模式；
- 4. 客户端模式。



- 1. Resource Owner 与 Client 之间，资源所有者向Client发起认证请求，Client再返回认证授权信息。
- 2. Client 收到 Resource Owner 的认证请求后，会去Authorization Server 申请访问令牌，Authorization Server会让Client 进行认证，通过之后会返回Access Token。
- 3. Client 拿到 Authorization Server 的 Access Token，访问Resource Server，Resource Server 验证之后，返回被保护的资源信息。
- 4. Resource Server 可以通过JWT在本地进行验证，也可以访问 Authorization Server，对Client 的请求的合法性进行验证。

3.2.4 OAuth2 授权码模式



1. 客户端携带 `client_id`, `scope`, `redirect_uri`, `state` 等信息引导用户请求授权服务器的授权端点下发 `code`。
2. 授权服务器验证客户端身份，验证通过则询问用户是否同意授权（此时会跳转到用户能够直观看到的授权页面，等待用户点击确认授权）。
3. 假设用户同意授权，此时授权服务器会将 `code` 和 `state`（如果客户端传递了该参数）拼接在 `redirect_uri` 后面，以302(重定向)形式下发 `code`。
4. 客户端携带 `code`, `redirect_uri`, 以及 `client_secret` 请求授权服务器的令牌端点下发 `access_token`。
5. 授权服务器验证客户端身份，同时验证 `code`，以及 `redirect_uri` 是否与请求 `code` 时相同，验证通过后下发 `access_token`，并选择性下发 `refresh_token`，支持令牌的刷新。

示例：

1. 授权请求：

```
response_type=code           // 必选项
&client_id={客户端的ID}     // 必选项
&redirect_uri={重定向URI}    // 可选项
&scope={申请的权限范围}     // 可选项
&state={任意值}             // 可选项
```

2. 授权响应参数：

```
code={授权码}                // 必填
&state={任意文字}           // 如果授权请求中包含 state的话那就是必填
```

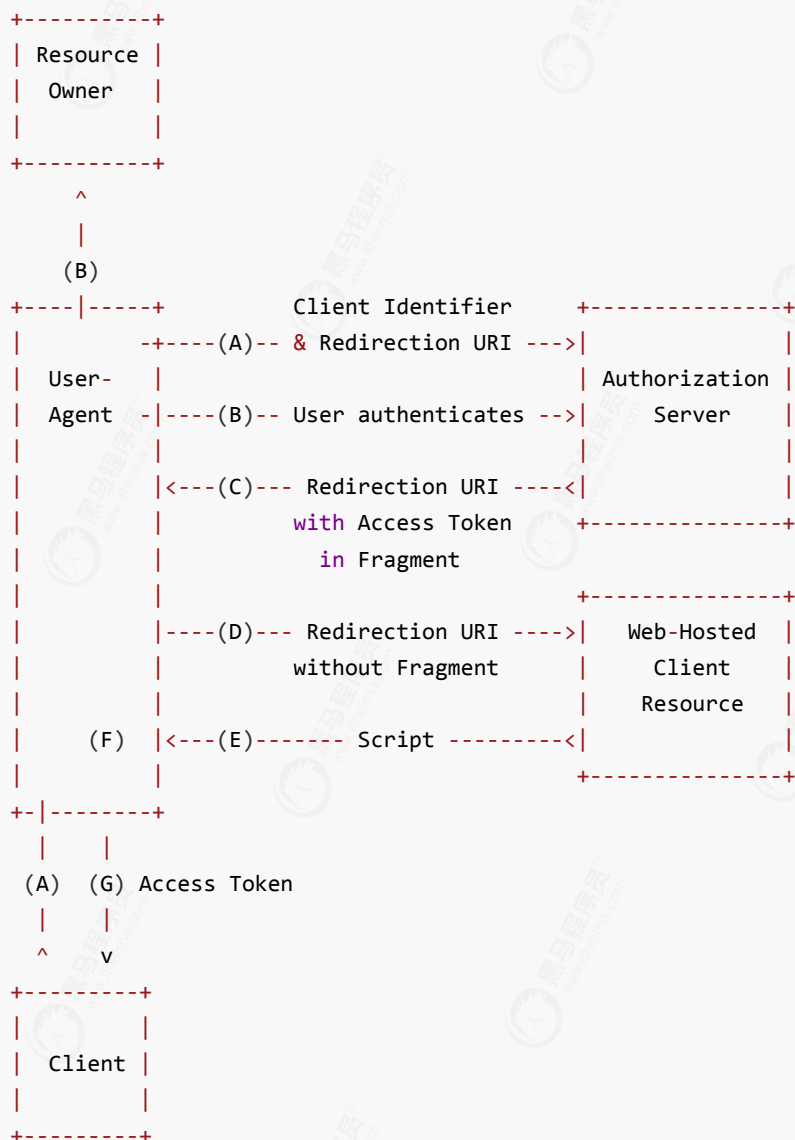
3. 令牌请求：

```
grant_type=authorization_code // 必填
&code={授权码}                // 必填 必须是认证服务器响应给的授权码
&redirect_uri={重定向URI}     // 如果授权请求中包含 redirect_uri 那就是必填
&code_verifier={验证码}      // 如果授权请求中包含 code_challenge 那就是必填
```

4. 令牌响应：

```
"access_token": "{访问令牌}", // 必填
"token_type": "{令牌类型}",   // 必填
"expires_in": {过期时间},     // 任意
"refresh_token": "{刷新令牌}", // 任意
"scope": "{授权范围}"         // 如果请求和响应的授权范围不一致就必填
```

3.2.5 OAuth2 隐式/简化模式



1. 资源所有者（用户）通过代理（WEB浏览器）访问客户端程序，发起简化模式认证。
2. 客户端（Client）向认证服务器（Auth Server）发起请求，此时客户端携带了客户端标识（client_id）和重定向地址（redirect_uri）。
3. 客户端（Client）拿到令牌 token 后就可以向第三方的资源服务器请求资源了。

示例:

1. 授权请求:

```
response_type=token           // 必选项
&client_id={客户端的ID}      // 必选项
&redirect_uri={重定向URI}    // 可选项
&scope={申请的权限范围}      // 可选项
&state={任意值}              // 可选项
```

2. 授权响应参数:

```

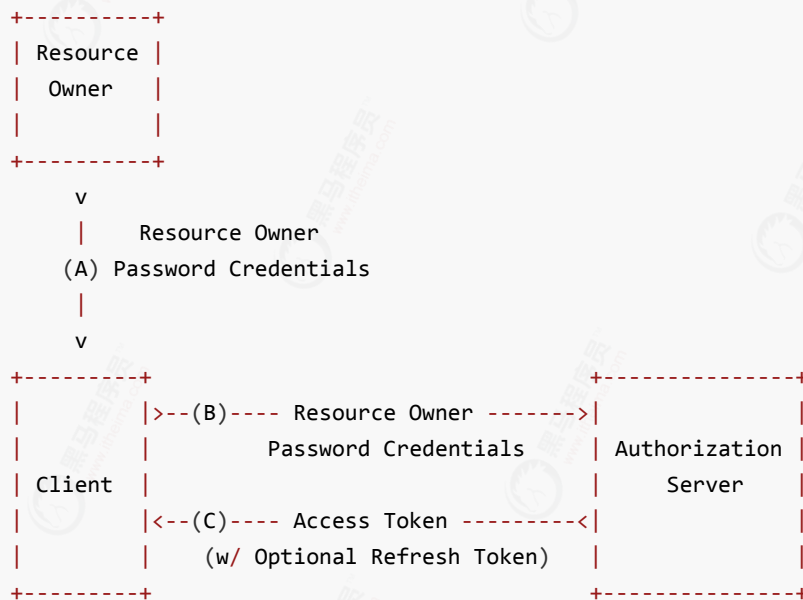
&access_token={令牌信息}    // 必填
&expires_in={过期时间}      // 任意
&state={任意文字}           // 如果授权请求中包含 state 那就是必填
&scope={授权范围}           // 如果请求和响应的授权范围不一致就必填

```

思考：为什么要有授权码和简化模式？看完这两种模式，可能会有些疑问，为什么要这么麻烦，直接一次请求返回**TOKEN**不就可以吗？

我们可以看出，两者主要差别，是少了code验证环节，直接返回token了，code验证是客户端与认证服务器在后台进行请求获取，代理是获取不到**TOKEN**的，如果缺少这个环节，直接返回**TOKEN**，相当于直接暴露给所有参与者，存在安全隐患，所以简化模式，一般用于信赖度较高的环境中使用。

3.2.6 OAuth2 密码模式



1. 资源拥有者直接通过客户端发起认证请求。
2. 客户端提供用户名和密码，向认证服务器发起请求认证。
3. 认证服务器通过之后，客户端（Client）拿到令牌 token 后就可以向第三方的资源服务器请求资源了。

示例：

1. 令牌请求：

```

grant_type=password          // 必填
&username={用户ID}          // 必填
&password={密码}             // 必填
&scope={授权范围}           // 任意

```

2. 令牌响应：

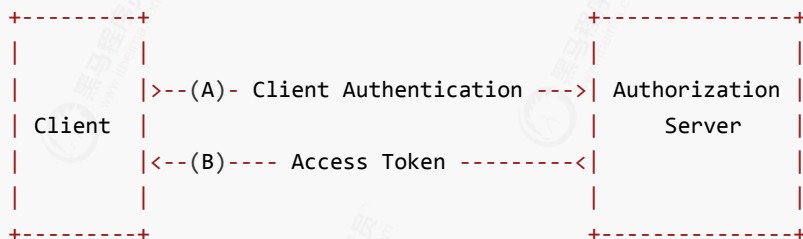
```

"access_token": "{访问令牌}", // 必填
"token_type": "{令牌类型}", // 必填
"expires_in": "{过期时间}", // 任意
"refresh_token": "{刷新令牌}", // 任意
"scope": "{授权范围}" // 如果请求和响应的授权范围不一致就必填

```

此模式简化相关步骤，直接通过用户和密码等隐私信息进行请求认证，认证服务器直接返回token，这需要整个环境具有较高的安全性。

3.2.7 OAuth2 客户端模式



1. 此模式最为简单直接，由客户端直接发起请求。
2. 客户端与服务器信赖度较高，服务端根据请求直接认证返回token信息。
3. 客户端（Client）拿到令牌 token 后就可以向第三方的资源服务器请求资源了。

这种模式一般在内部服务之间应用，授权一次，长期可用，不用刷新token。

示例:

1. 令牌请求:

```

grant_type=client_credentials // 必填
client_id={客户端的ID} // 必填
client_secret={客户端的密钥} // 必填
&scope={授权范围} // 任意

```

2. 令牌响应:

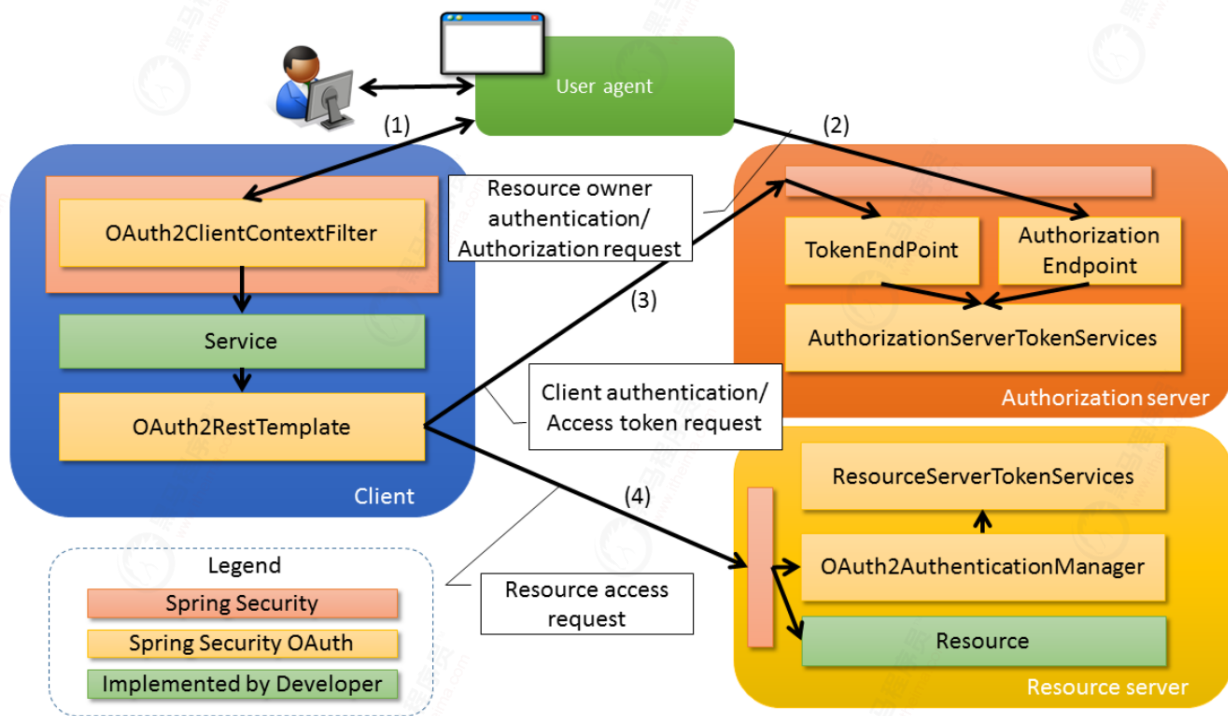
```

"access_token": "{访问令牌}", // 必填
"token_type": "{令牌类型}", // 必填
"expires_in": "{过期时间}", // 任意
"scope": "{授权范围}" // 如果请求和响应的授权范围不一致就必填

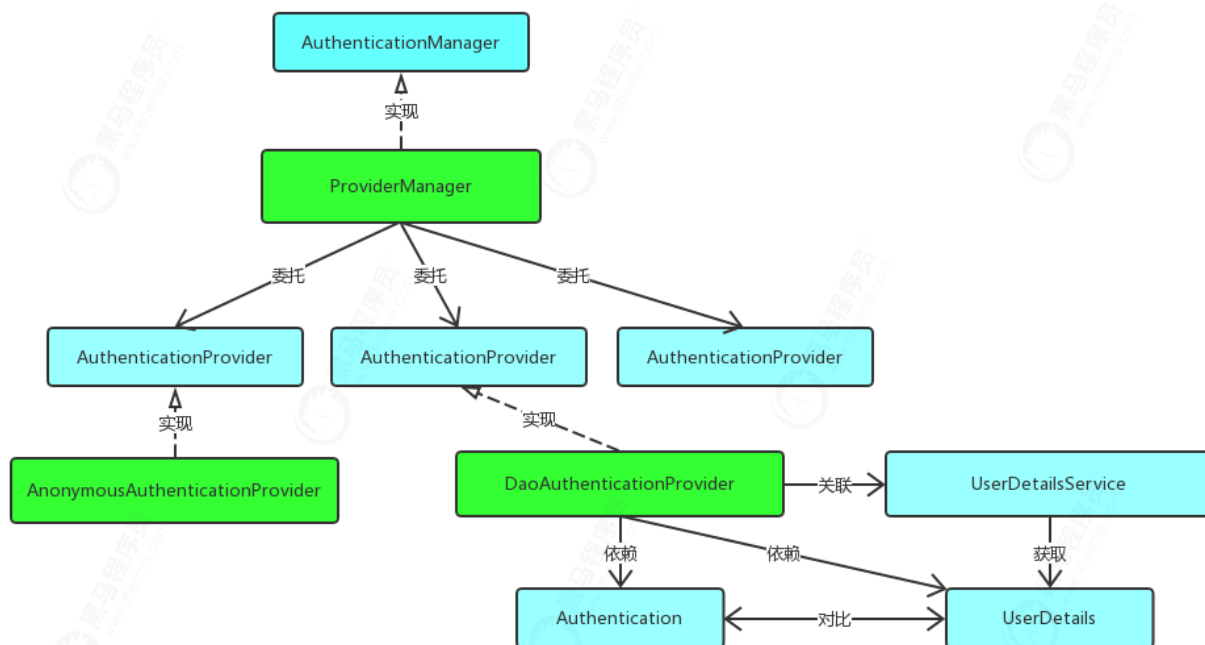
```

3.2.8 Spring Security OAuth设计

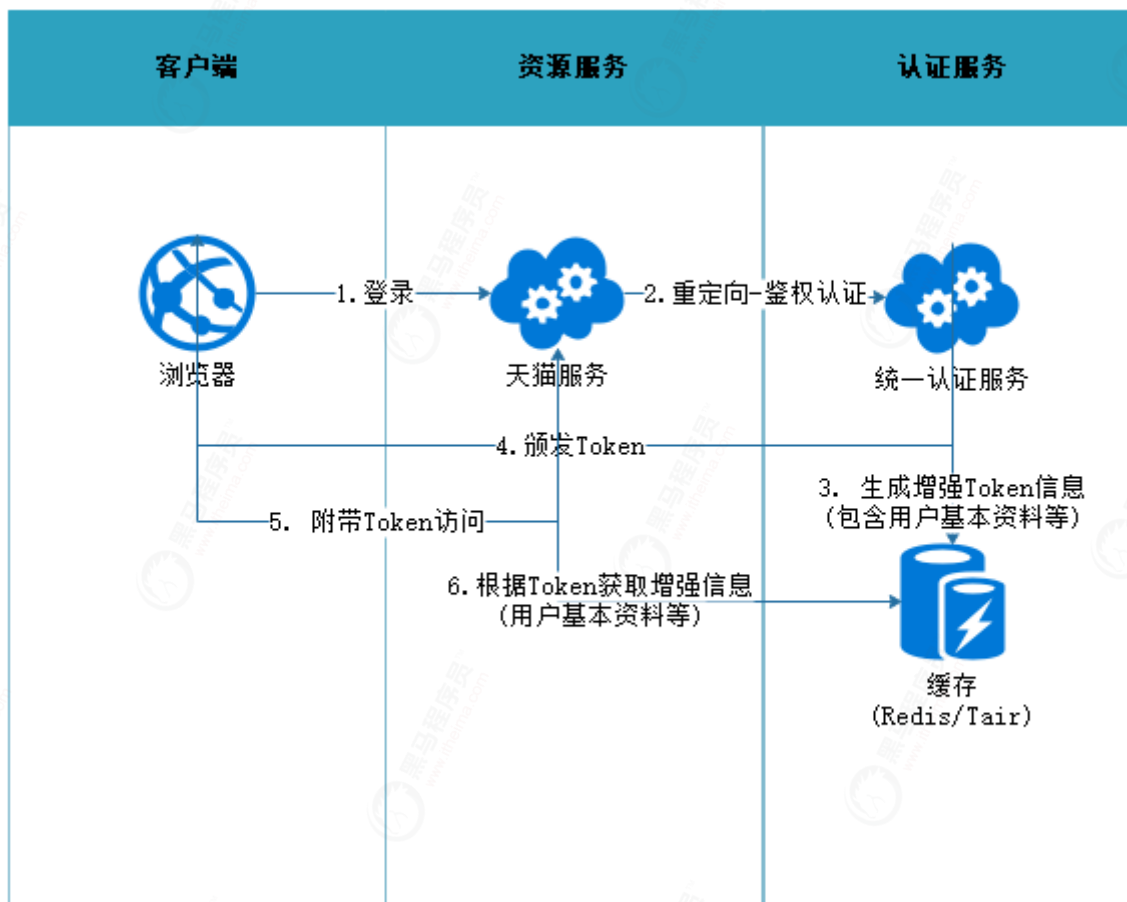
整体设计结构:



UML类图:



3.2.9 增强Token技术解决方案



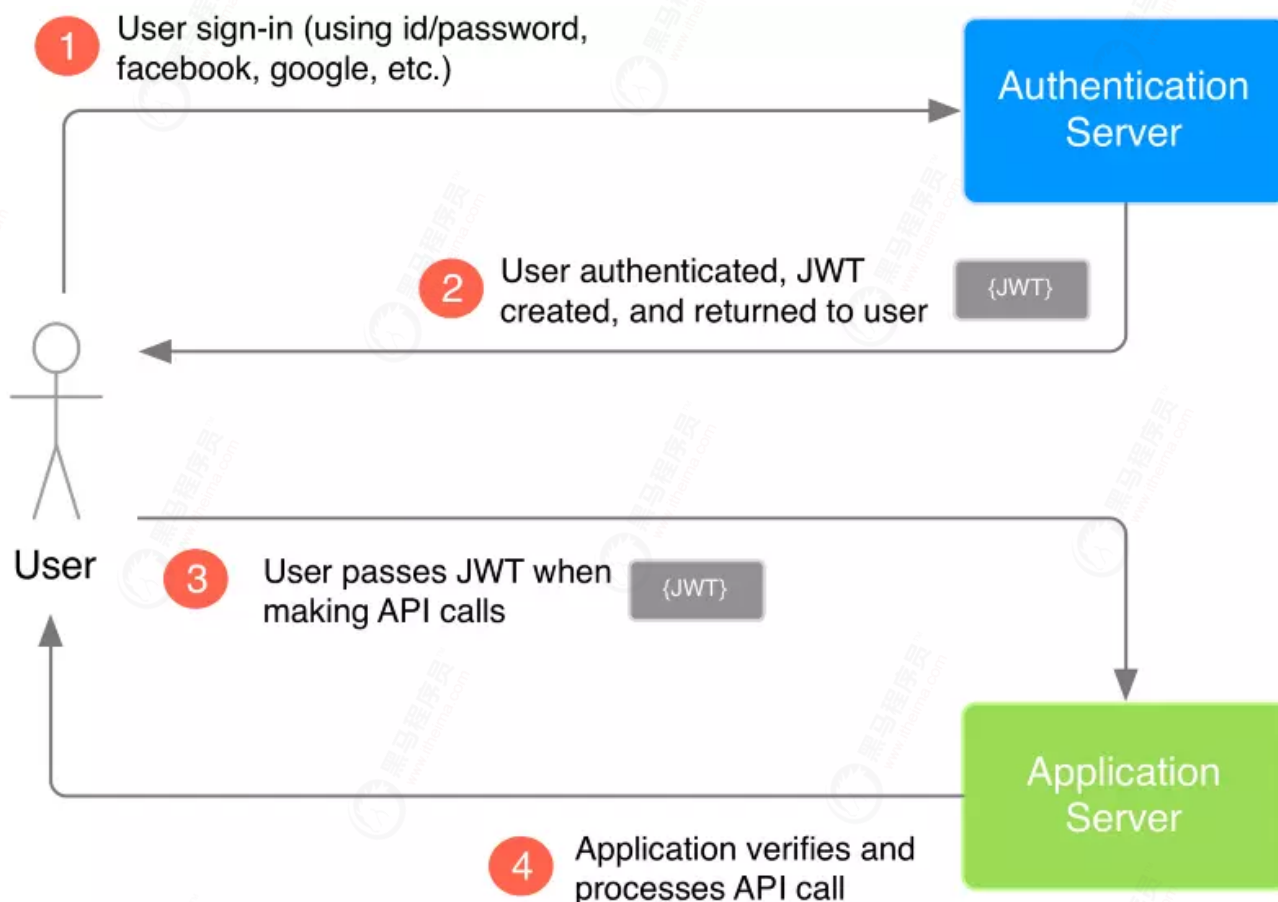
优势与应用场景

基于Token的鉴权方案，实现方式有多种，增强Token属于其中一种，为什么要采用增强Token方式，它能够解决怎样的问题？普通Token认证方式，没有附带必要的用户信息，如果要查询，需要再次调用OAuth2的用户资料认证接口，会增加传输开销；JWT虽然能够附带一定用户信息，但受限于长度，存储空间有限；如果既要保障性能，又要求能够存储一定的信息，就可以采用增强Token方案，它是将信息存储至Redis缓存中，作为资源服务，接收到Token之后，可以直接从Redis中获取信息。

它可以适用于微服务架构下，有一定用户信息要求的场景，比如订单服务、资金服务需要获取用户的基本资料，但如果是跨IDC，跨区域，需要暴露外网的情况下，不推荐采用此方案，因为需要保障数据的安全性。

3.2.10 JWT技术解决方案

JWT认证流程：



JWT应用场景:

1. 认证 Authentication;
2. 授权 Authorization // 注意这两个单词的区别;
3. 联合识别;
4. 客户端会话（无状态的会话）;
5. Restful Api 无状态认证。

JWT缺陷:

1. 更多的空间占用。如果将存在服务端session中的各类信息都放在JWT中保存在客户端，可能造成JWT占用更大空间，需要考虑cookie的空间限制因素，如果放在Local Storage，则可能受到XSS攻击。
2. 更不安全。这里是特指将JWT保存在Local Storage中，然后使用Javascript取出后作为HTTP header发送给服务端的方案。在Local Storage中保存敏感信息并不安全，容易受到跨站脚本攻击，跨站脚本（Cross site script，简称xss）是一种“HTML注入”，由于攻击的脚本多数时候是跨域的，所以称之为“跨域脚本”，这些脚本代码可以盗取cookie或是Local Storage中的数据([XSS攻击](#)的原理解释)。
3. 无法作废已颁布的令牌。所有的认证信息都在JWT中，由于在服务端是无状态，即使你知道了某个JWT被盗取了，也没有办法将其作废。在JWT过期之前，除非主动增加过期接口，否则无法处理。
4. 续签问题。传统 session请求时是可以自动续期，payload之中有一个exp过期时间参数，它可以代表JWT的时效性，但JWT自身设计并没有考虑续签问题，因为payload是参与签名处理，如果exp过期时间被修改，那整个JWT串就会产生变化，所以JWT原生并不支持续签。

JWT应用优化方案:

1. 针对安全性问题：可以使用Cookie存储，并设置HttpOnly=true，只能由服务端保存以及通过自动回传的cookie取得JWT，以便防御XSS攻击；在JWT载体中加入一个随机值作为CSRF令牌，服务端将令牌也保存在

Cookie中，前端可以取得该令牌并在请求时作为HTTP header头部信息传递，服务端在认证时，从JWT取出CSRF令牌和HEADER中的令牌做比对，从而防止CSRF的攻击。

- 续签问题：通过Token的Refresh机制来实现，需要对JWT的传递做统一封装，客户端再开辟一个线程定期检测有效期，临近过期时重新刷新Tokens，进行全局更新。

JWT扩展知识:

- JWS(JSON Web Signature):** 其结构就是在JWT的基础上，在头部声明签名算法，并在最后添加上签名。创建签名，是保证jwt不能被他人随意篡改。为了完成签名，除了用到header信息和payload信息外，还需要算法的密钥，也就是secret。当利用非对称加密方法的时候，这里的secret就是为私钥。
- JWE(JSON Web Encryption):** 它能够保护数据不被第三方查看，JWT是通过签名来验证数据来源的合法性，但载体信息只是通过Base64编码，不能严格保障数据的安全性，通过JWE，能够使JWT变得更为安全。

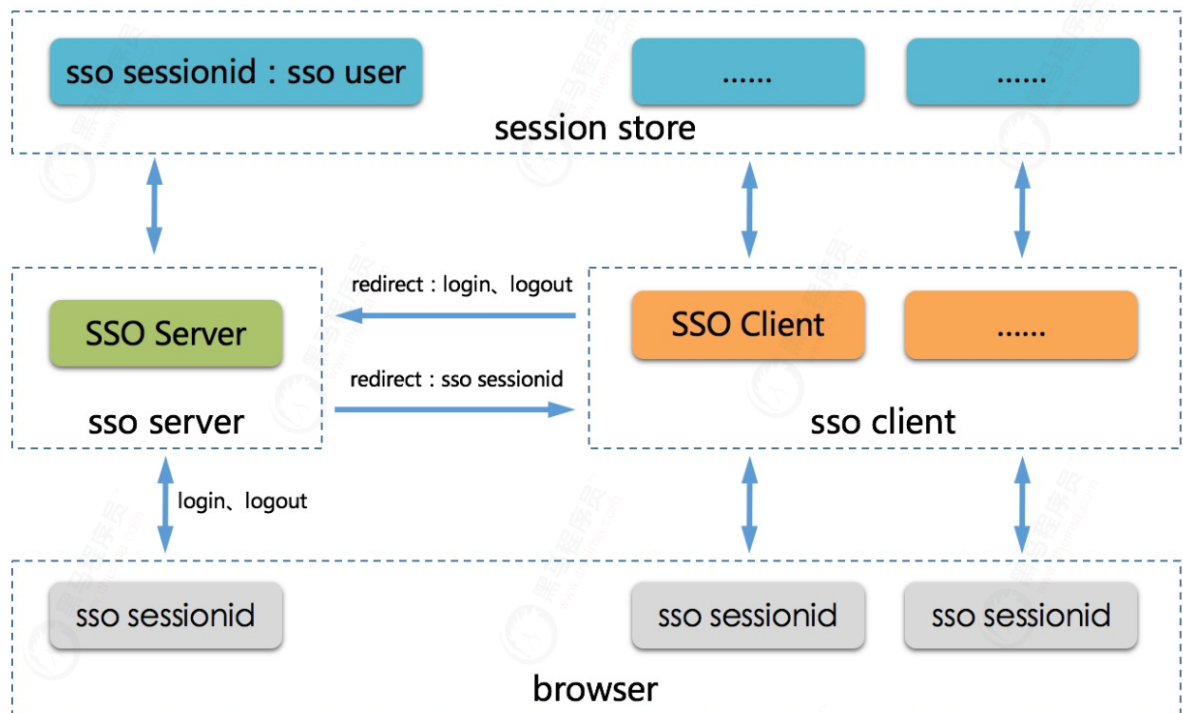
JWE数据组成结构:



4. 单点登录之生产实践

4.1 基于Cookie跨域与分布式Session的技术实践

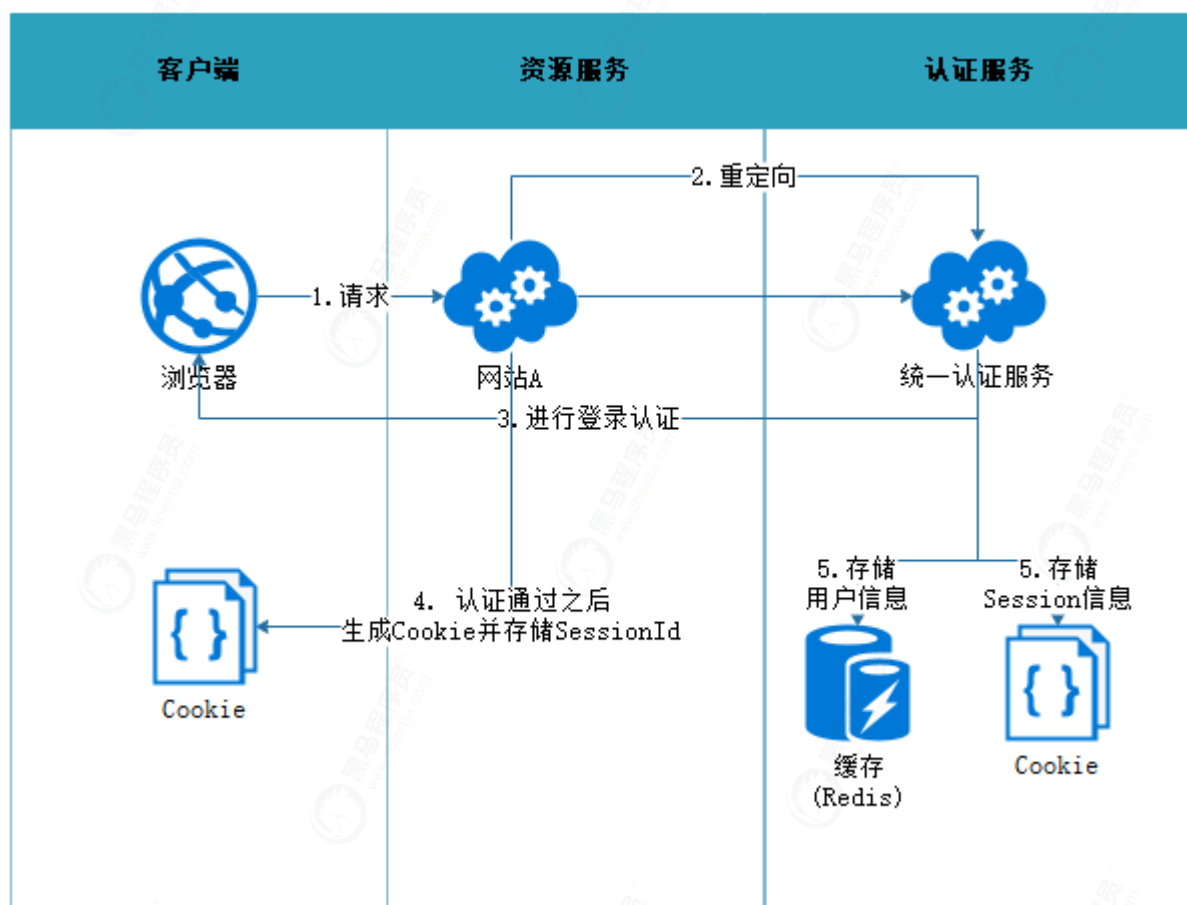
- XXL-SSO整体架构:



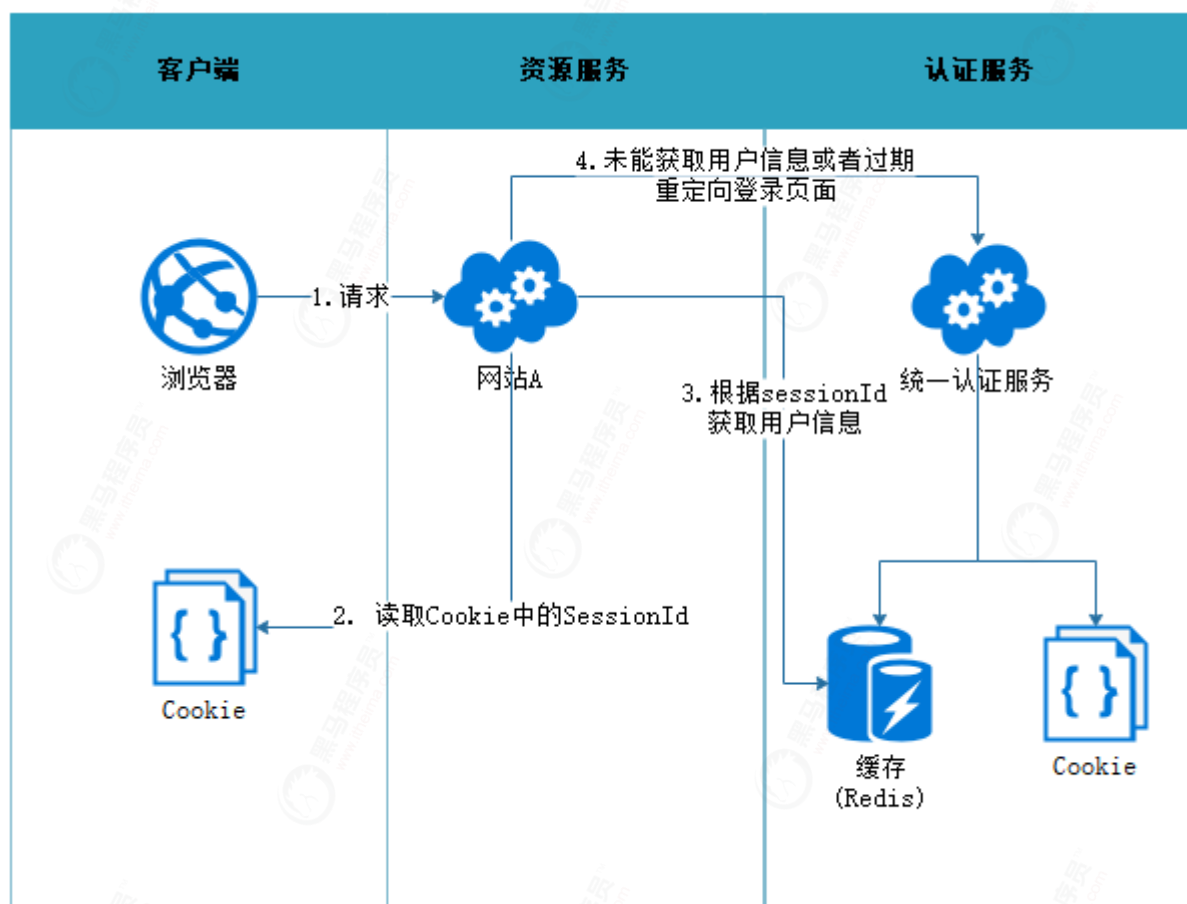
XXL-SSO架构图 v0.1

- 实现原理剖析:

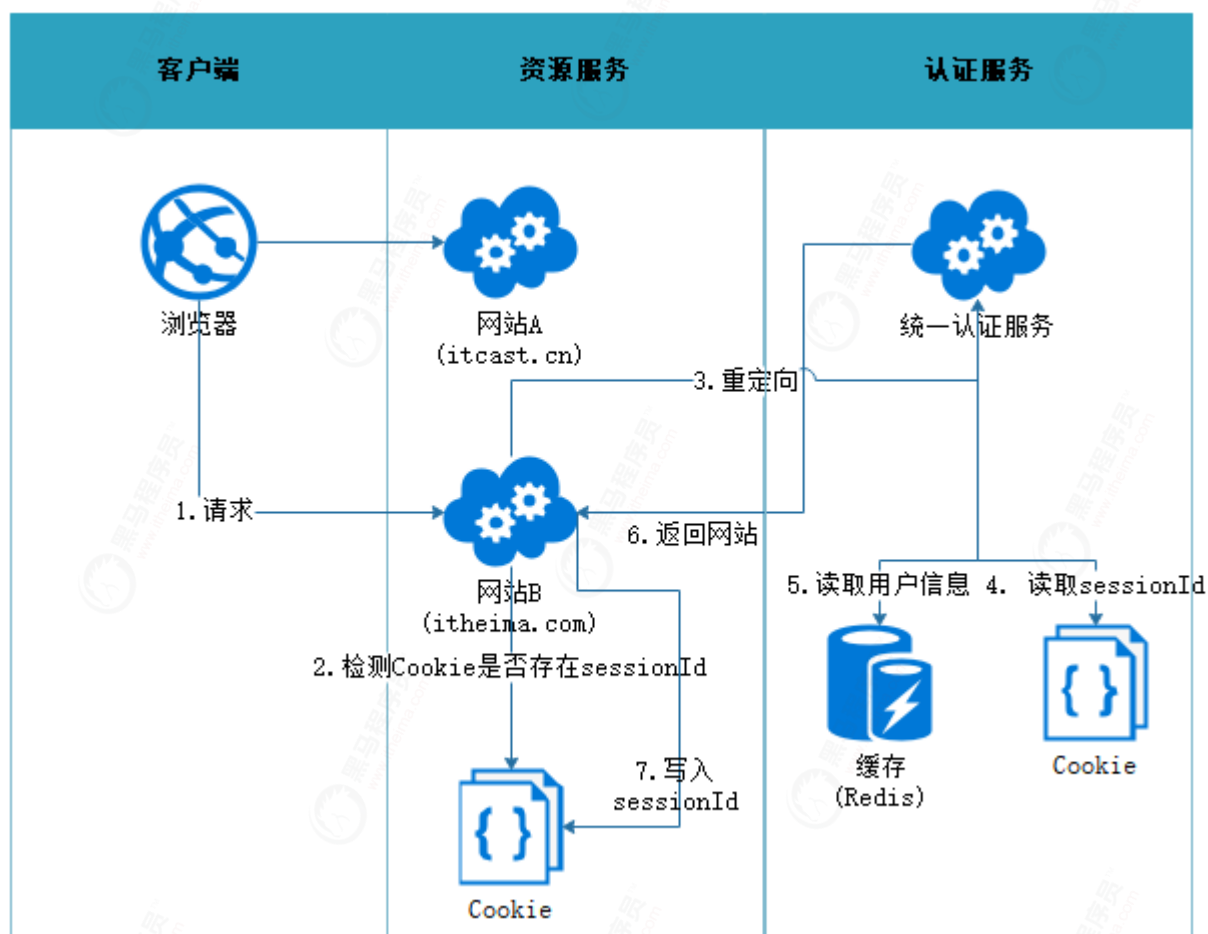
- 首次请求:



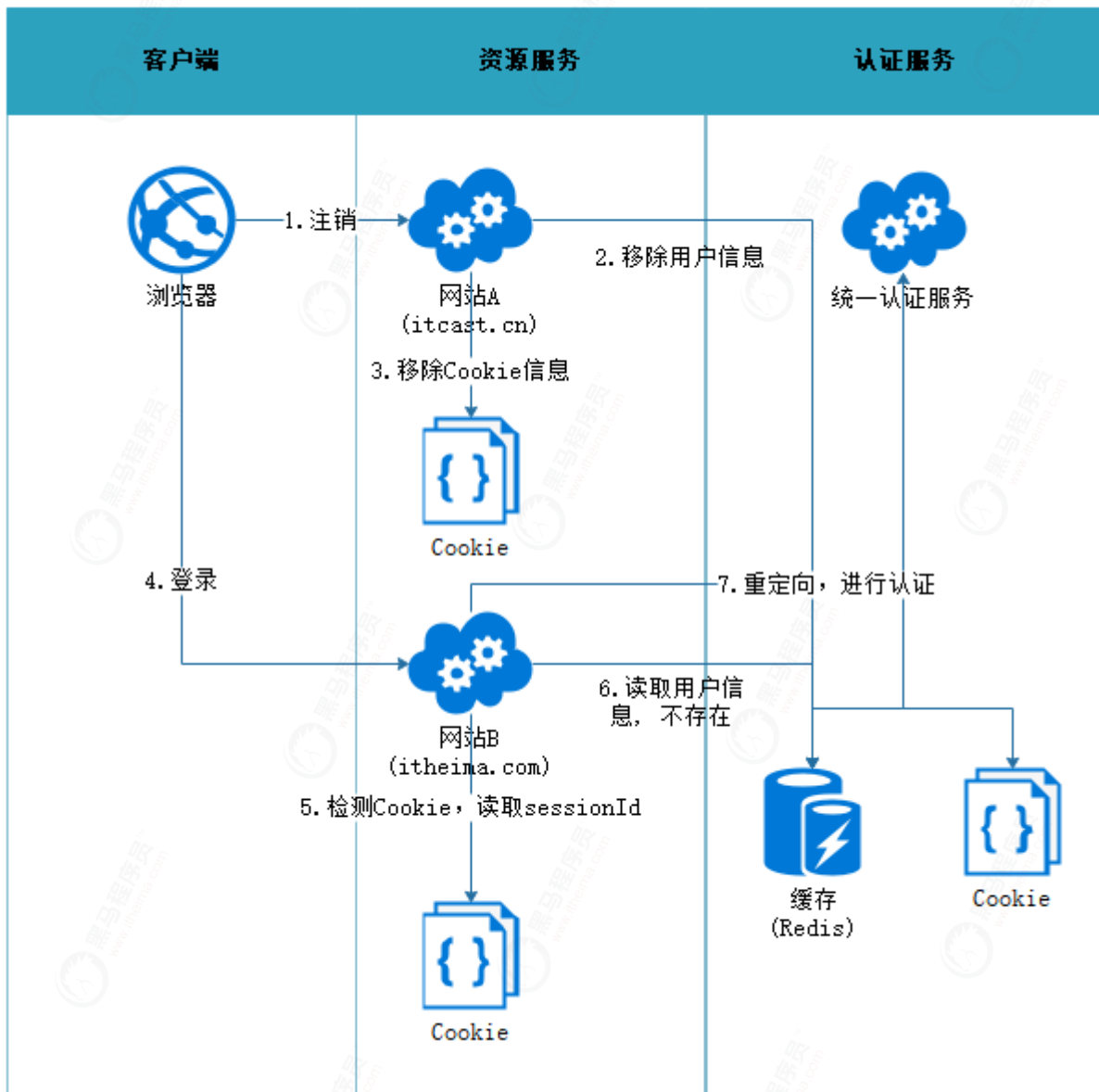
- 第二次请求



- 跨域请求



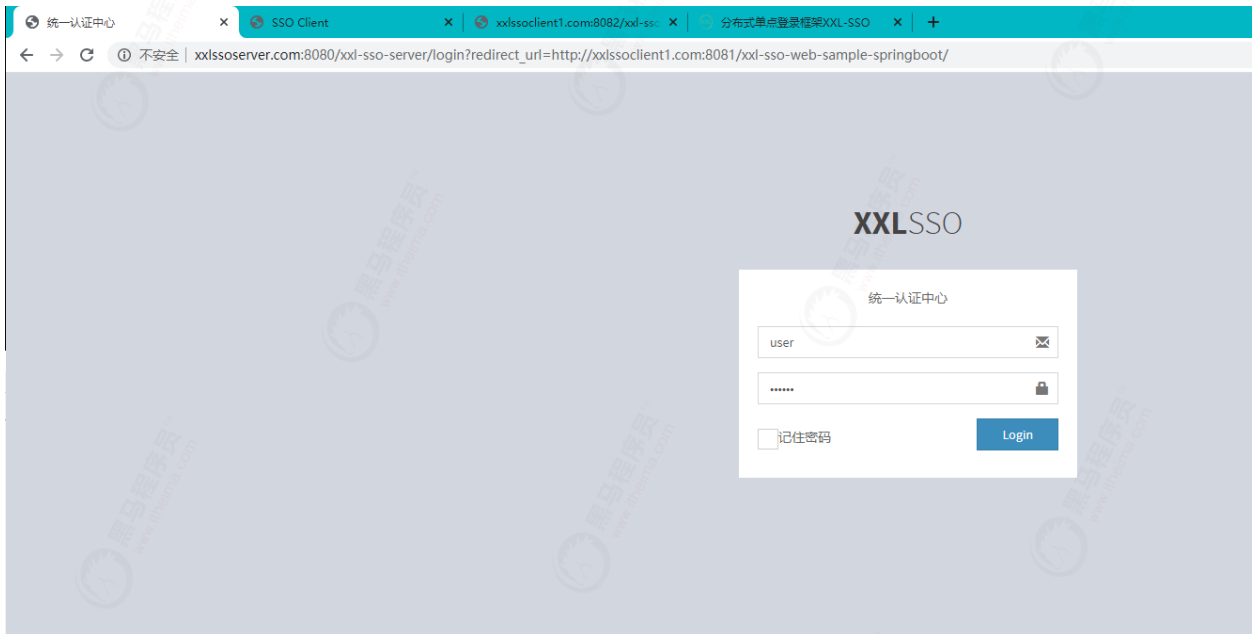
- 注销流程



3. 实例演示:

- 首次登陆跳转至统一认证中心

访问: <http://xxlssoclient1.com:8081/>



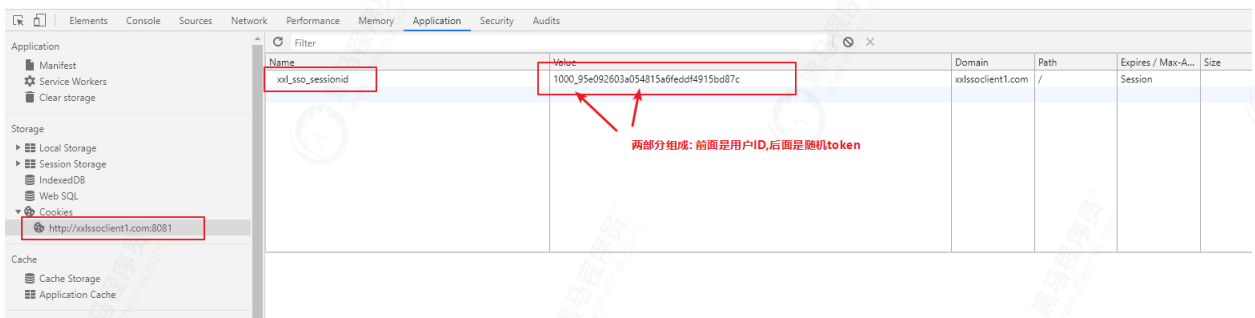
- 登陆成功

写入Cookie，保存sessionId信息。



[user] login success.

[Logout](#)



- 跨域访问

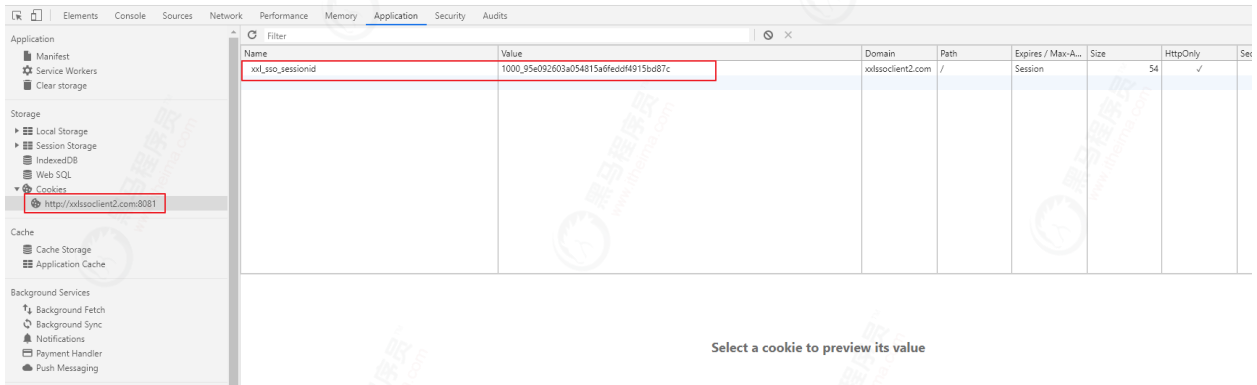
访问另外一个域名: <http://xxlssoclient2.com:8081/>

自动登陆，并且写入SessionId至Cookie当中。



[user] login success.

[Logout](#)



4. 代码实现剖析:

采用Debug方式跟踪解析。

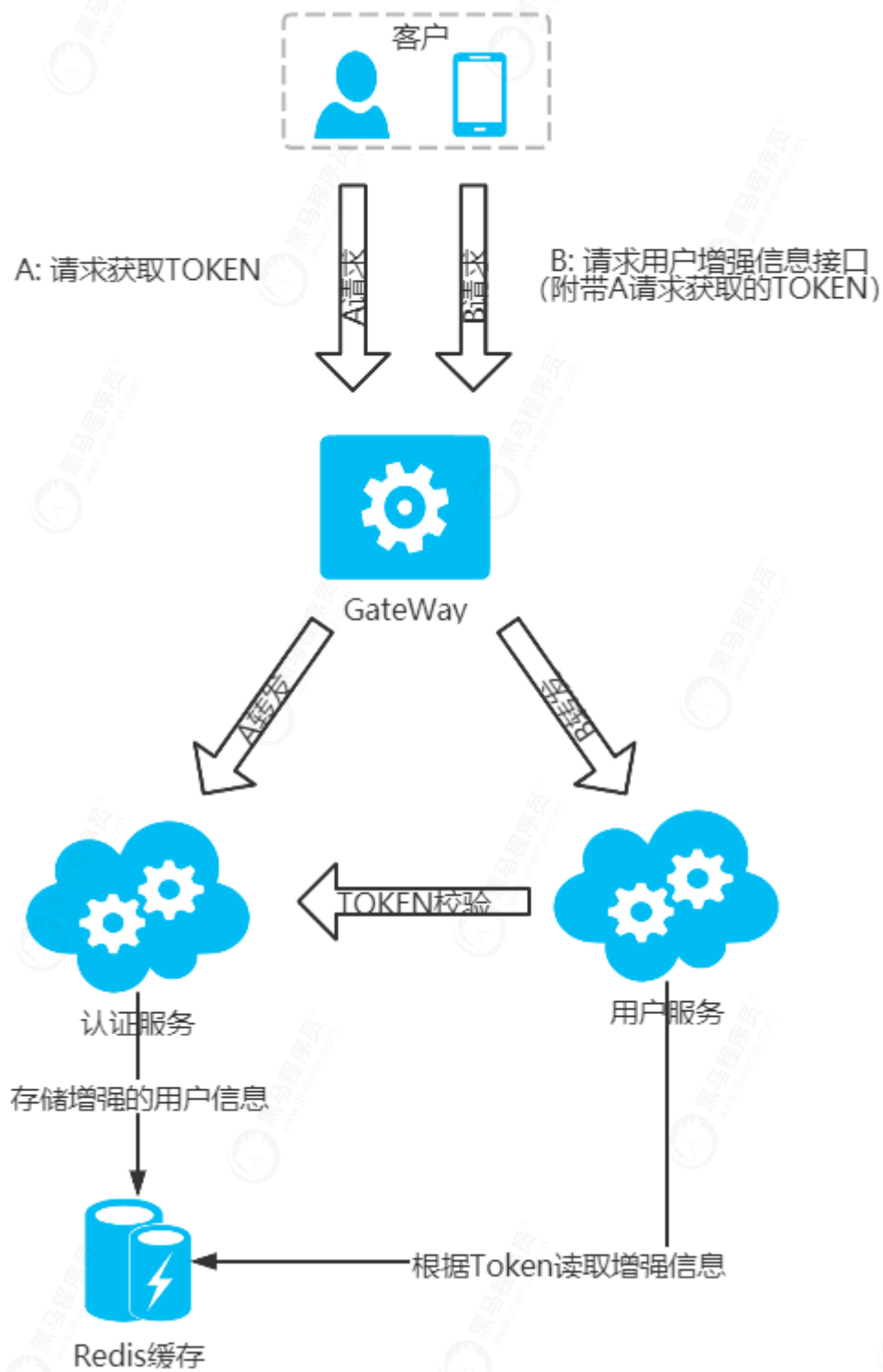
关键断点:

- 统一认证服务, 登陆入口:
WebController的login方法
- 应用服务Web过滤器:
XxlSsoWebFilter的doFilter方法
- 应用服务登陆注销:
XxlSsoWebFilter的doFilter方法, Line: 64。

4.2 基于Token增强的微服务技术实践

1. 整体实现流程

采用密码模式, 基于Token增强的微服务应用实现方案:



2. 代码实现

认证服务

- 认证服务配置

AuthorizationServerConfig

```

...
/**
 * 自定义Client查询，可以修改表名， 字段等
 * @param clients
 */
@Override
@SneakyThrows
public void configure(ClientDetailsServiceConfigurer clients) {
    AuthClientDetailService clientDetailsService = new
AuthClientDetailService(dataSource);
    clientDetailsService.setSelectClientDetailsSql(DEFAULT_SELECT_STATEMENT);
    clientDetailsService.setFindClientDetailsSql(DEFAULT_FIND_STATEMENT);
    clients.withClientDetails(clientDetailsService);
}

/**
 * 防止申请token时出现401错误
 * @param oauthServer
 */
@Override
public void configure(AuthorizationServerSecurityConfigurer oauthServer) {
    oauthServer
        .tokenKeyAccess("permitAll()")
        .checkTokenAccess("permitAll()")
        .allowFormAuthenticationForClients();
}

/**
 * 认证服务配置
 * @param endpoints
 */
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints
        .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST)
        .tokenStore(tokenStore())
        .tokenEnhancer(tokenEnhancer())
        .userDetailsService(authStockUserDetailService)
        .authenticationManager(authenticationManager)
        .reuseRefreshTokens(false);
}

/**
 * TokenStore实现方式， 采用Redis缓存
 * @return
 */
@Bean
public TokenStore tokenStore() {
    RedisTokenStore tokenStore = new RedisTokenStore(redisConnectionFactory);
    tokenStore.setPrefix(GlobalConstants.OAUTH_PREFIX_KEY);

    tokenStore.setAuthenticationKeyGenerator(new

```

```

DefaultAuthenticationKeyGenerator() {
    @Override
    public String extractKey(OAuth2Authentication authentication) {
        return super.extractKey(authentication);
    }
});
return tokenStore;
}

/**
 * token增强处理， 支持扩展信息
 * @return TokenEnhancer
 */
@Bean
public TokenEnhancer tokenEnhancer() {
    return (accessToken, authentication) -> {
        try {
            if (GlobalConstants.OAUTH_CLIENT_CREDENTIALS
                .equals(authentication.getOAuth2Request().getGrantType())) {
                return accessToken;
            }
            // 通过MAP 存储附加的信息
            final Map<String, Object> additionalInfo = new HashMap<>(16);
            OAuthTradeUser authTradeUser = (OAuthTradeUser)
authentication.getUserAuthentication().getPrincipal();
            if (null != authTradeUser) {
                TradeUser tradeUser = authTradeUser.getTradeUser();
                // 需要扩充增加的用户附带信息
                additionalInfo.put(GlobalConstants.OAUTH_DETAILS_USER_ID,
tradeUser.getId());
                additionalInfo.put(GlobalConstants.OAUTH_DETAILS_USERNAME,
tradeUser.getName());
                additionalInfo.put(GlobalConstants.OAUTH_DETAILS_LOGIN_INFO,
tradeUser.getEmail() + "|" + tradeUser.getAddress());
                additionalInfo.put("active", true);
            }
            // 将附加的信息记录保存， 形成增强的TOKEN
            ((DefaultOAuth2AccessToken)
accessToken).setAdditionalInformation(additionalInfo);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        }
        return accessToken;
    };
}
}

```

- 用户信息服务接口

AuthStockUserDetailServiceImpl

```

@Service("authStockUserDetailsService")
public class AuthStockUserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private TradeUserRepository tradeUserRepository;

    @Autowired
    private CacheManager cacheManager;

    @Override
    public UserDetails loadUserByUsername(String userNo) throws UsernameNotFoundException {

        // 查询缓存
        Cache cache = cacheManager.getCache(GlobalConstants.OAUTH_KEY_STOCK_USER_DETAILS);
        if (cache != null && cache.get(userNo) != null) {
            return (UserDetails) cache.get(userNo).get();
        }
        // 缓存未找到， 查询数据库
        TradeUser tradeUser = tradeUserRepository.findByUserNo(userNo);
        if(null == tradeUser){
            throw new UsernameNotFoundException(userNo + " not valid !");
        }
        // 封装成OAUTH鉴权的用户对象
        UserDetails userDetails = new OAuthTradeUser(tradeUser);
        // 将用户信息放入缓存
        cache.put(userNo, userDetails);
        return userDetails;
    }
}

```

这是Spring Security 提供的用户信息接口，采用OAUTH的密码模式， 需要实现该接口的loadUserByUsername方法，为提升性能， 这里我们加入了Spring Cache缓存处理。

- 自定义用户信息：
OAuthTradeUser

```

public class OAuthTradeUser extends User {

    private static final long serialVersionUID = -1L;

    /**
     * 业务用户信息
     */
    private TradeUser tradeUser;

    public OAuthTradeUser(TradeUser tradeUser) {
        // OAUTH2认证用户信息构造处理
        super(tradeUser.getUserNo(), tradeUser.getUserPwd(), (tradeUser.getStatus() == 0 ?
true : false),
            true, true, (tradeUser.getStatus() == 0 ? true : false),
Collections.emptyList());
        this.tradeUser = tradeUser;
    }

}

```

- 客户端信息服务接口

AuthClientDetailsService

```

public class AuthClientDetailsService extends JdbcClientDetailsService {

    public AuthClientDetailsService(DataSource dataSource) {
        super(dataSource);
    }

    /**
     * 重写原生方法支持redis缓存
     *
     * @param clientId
     * @return
     * @throws InvalidClientException
     */
    @Override
    @Cacheable(value = GlobalConstants.OAUTH_KEY_CLIENT_DETAILS, key = "#clientId", unless
= "#result == null")
    public ClientDetails loadClientByClientId(String clientId) {
        return super.loadClientByClientId(clientId);
    }

}

```

这是OAUTH内置的客户端信息， 重新它是为了实现缓存， 减少数据库查询。

用户服务

- 认证配置ResourceSecurityConfigurer

```

@Primary
@Order(90)
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceSecurityConfigurer implements ResourceServerConfigurer {

    @Autowired
    protected RemoteTokenServices remoteTokenServices;

    @Autowired
    private RestTemplate lbRestTemplate;

    /**
     * 远程调用, 采用restTemplate方式处理
     * @param resourceServerSecurityConfigurer
     * @throws Exception
     */
    @Override
    public void configure(ResourceServerSecurityConfigurer
resourceServerSecurityConfigurer) throws Exception {
        remoteTokenServices.setRestTemplate(lbRestTemplate);
        resourceServerSecurityConfigurer.tokenServices(remoteTokenServices);
    }

    /**
     * 资源服务安全配置
     * @param httpSecurity
     * @throws Exception
     */
    @Override
    public void configure(HttpSecurity httpSecurity) throws Exception {

        httpSecurity.csrf().disable()
            .authorizeRequests()
            .antMatchers("/user/**").authenticated().and()
            .formLogin().loginPage("/login")
            .failureUrl("/login?error")
            .defaultSuccessUrl("/home");
    }

    /**
     * RestTemplate配置
     * @return
     */
    @Bean
    @Primary
    @LoadBalanced
    public RestTemplate lbRestTemplate() {
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.setErrorHandler(new DefaultResponseErrorHandler() {

            @Override

```

```

        public void handleError(ClientHttpResponse response) throws IOException {
            if (response.getRawStatusCode() != HttpStatus.BAD_REQUEST.value()) {
                super.handleError(response);
            }
        }
    });
    return restTemplate;
}
}

```

用户服务为资源服务，认证采用RestTemplate调用方式。资源服务一定要开启@EnableResourceServer注解，@EnableGlobalMethodSecurity为方法级别安全控制。

- 提供获取用户增强信息接口

StockUserController

```

/**
 * 获取用户增强信息
 * @param userNo
 * @param userPwd
 * @return
 */
@RequestMapping("/getUserEnhancer")
public ApiResponseResult getUserEnhancer() {

    ApiResponseResult result = null;
    try {
        // 用户登陆逻辑处理
        Map<String, Object> userAdditionalInfos = getUserAdditionalInfos();
        result = ApiResponseResult.success(userAdditionalInfos);
    } catch (ComponentException e) {
        log.error(e.getMessage(), e);
        result = ApiResponseResult.error(e.getErrorCodeEnum());
    } catch (Exception e) {
        log.error(e.getMessage(), e);
        result = ApiResponseResult.sysError(e.getMessage());
    }

    return result;
}
}

```

网关服务:

- 全局过滤器StockRequestGlobalFilter

```

@Component
@Log4j2
public class StockRequestGlobalFilter implements GlobalFilter, Ordered {
    private static final String HEADER_NAME = "X-Forwarded-Prefix";

    /**
     * 通过filter来自定义配置转发信息
     * @param exchange
     * @param chain
     * @return
     */
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        String authentication =
exchange.getRequest().getHeaders().getFirst("Authorization");
        if(!StringUtil.isNullOrEmpty(authentication)){
            log.info("enter stockRequestGlobalFilter filter method: " + authentication);
            exchange.getRequest().mutate().header("Authorization",authentication);
        }
        return chain.filter(exchange.mutate().build());
    }

    @Override
    public int getOrder() {
        return -1000;
    }
}

```

这是自定义全局过滤器的实现，防止header中的Authorization没有转发的的问题。

3. 测试验证

- o 申请Token

```

POST http://127.0.0.1:10680/oauth/token?
grant_type=password&username=admin&password=admin&scope=server
Accept: */*
Cache-Control: no-cache
Authorization: Basic YXBwOmFwcA==

```

返回Token信息:

```
{
  "access_token": "cc5c4c1d-b519-458f-b338-ad4bd1ec06b0",
  "token_type": "bearer",
  "refresh_token": "86fec4ff-6c24-4171-a257-bf2d4e6bc30c",
  "expires_in": 29749,
  "scope": "server",
  "login_info": "hekun1@itcast.cn|null",
  "user_id": 1,
  "user_name": "admin",
  "active": true
}
```

- 获取增强用户信息

```
GET 127.0.0.1:10680/user/getUserEnhancer
Accept: */*
Cache-Control: no-cache
Authorization: Bearer cc5c4c1d-b519-458f-b338-ad4bd1ec06b0
```

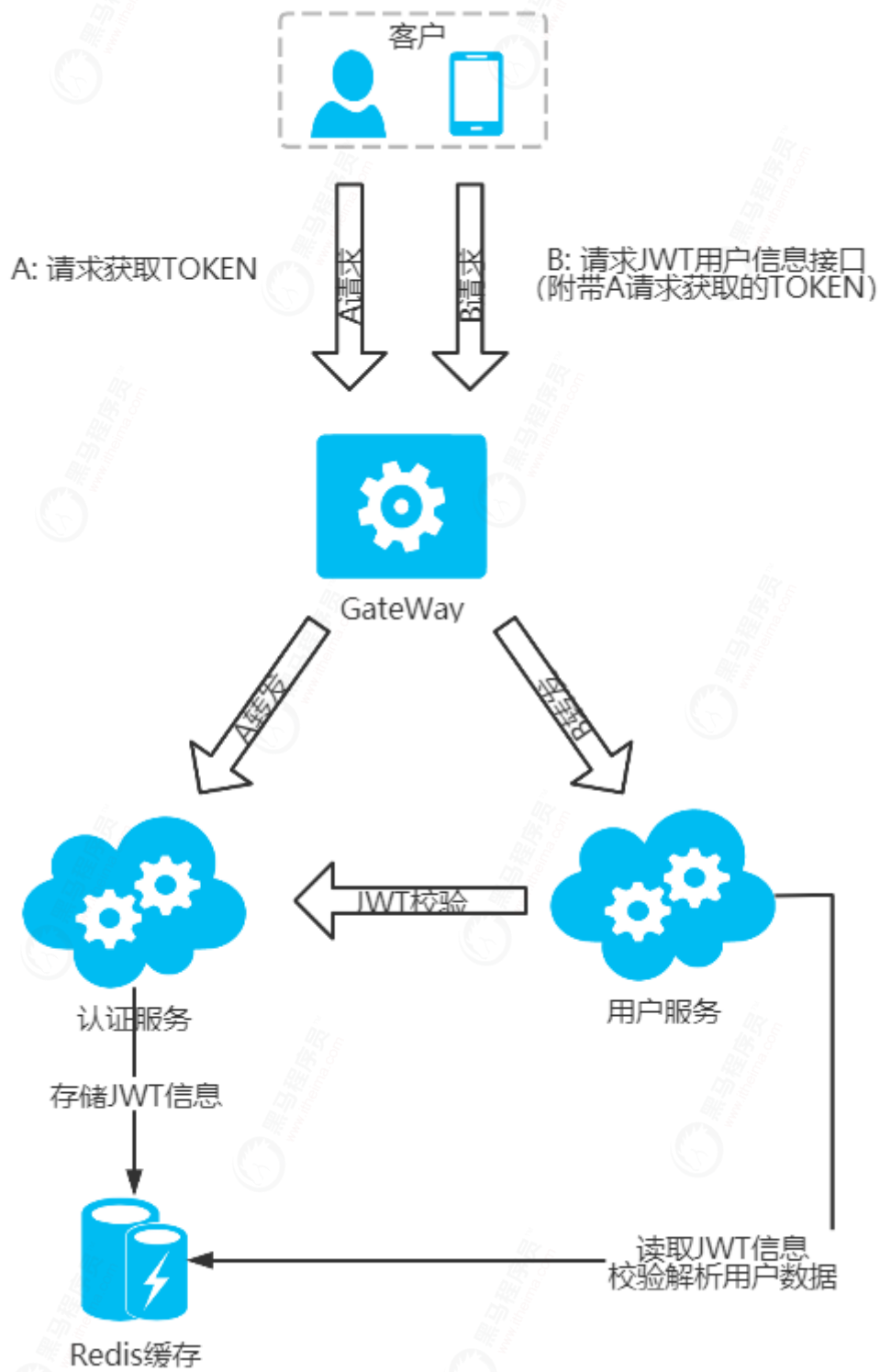
返回增强的用户信息:

```
{
  "code": "SYS_200",
  "msg": "成功",
  "extendData": null,
  "data": {
    "login_info": "hekun1@itcast.cn|null",
    "user_id": 1,
    "user_name": "admin",
    "active": true
  },
  "success": true
}
```

4.3 基于JWT扩展信息的微服务技术实践

1. 整体实现流程

采用密码模式，基于JWT扩展信息的微服务应用实践方案：



2. 代码实现

- 认证服务

- 认证服务系统配置

AuthorizationServerConfig

```

...
/**
 * 认证服务配置
 * @param endpoints
 */
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
    // JWT信息增强配置，采用链式配置， 包含JWT签名配置与JWT扩展信息配置。
    TokenEnhancerChain enhancerChain = new TokenEnhancerChain();
    List<TokenEnhancer> delegates = new ArrayList<>();
    delegates.add(jwtTokenEnhancer());
    delegates.add(accessTokenConverter());
    enhancerChain.setTokenEnhancers(delegates);

    endpoints
        .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST)
        .tokenStore(tokenStore())
        .userService(authStockUserDetailsService)
        .authenticationManager(authenticationManager)
        .reuseRefreshTokens(false)
        .tokenEnhancer(enhancerChain);
}

@Bean
public JwtTokenEnhancer jwtTokenEnhancer() {
    return new JwtTokenEnhancer();
}

/**
 * TokenStore实现方式， 采用Redis缓存
 * @return
 */
@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}

@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey("test123");
    return converter;
}
...

```

认证服务采用JWT方式配置，JWT配置，采用链式配置， 包含JWT签名配置与JWT扩展信息，JWT签名设为test123。这里采用自定义的增强JWT作实现。

- JWT增强实现类:

JwtTokenEnhancer:

```

public class JwtTokenEnhancer implements TokenEnhancer {

    /**
     * JWT扩展存储用户信息
     * @param accessToken
     * @param authentication
     * @return
     */
    @Override
    public OAuth2AccessToken enhance(OAuth2AccessToken accessToken,
    OAuth2Authentication authentication) {
        Map<String, Object> additionalInfo = new HashMap<>();
        OAuth2TradeUser authTradeUser = (OAuth2TradeUser)
authentication.getUserAuthentication().getPrincipal();
        if(null != authTradeUser) {
            TradeUser tradeUser = authTradeUser.getTradeUser();
            // 存储用户扩展信息
            additionalInfo.put(GlobalConstants.OAUTH_DETAILS_USER_ID,
tradeUser.getId());
            additionalInfo.put(GlobalConstants.OAUTH_DETAILS_USERNAME,
tradeUser.getName());
            additionalInfo.put(GlobalConstants.OAUTH_DETAILS_LOGIN_INFO,
tradeUser.getEmail() + "|" + tradeUser.getAddress());
            ((DefaultOAuth2AccessToken)
accessToken).setAdditionalInformation(additionalInfo);
        }
        return accessToken;
    }
}

```

在JWT存储扩展用户信息，可以根据需要扩展不同的信息，但长度要有限制。

- 用户服务
 - 认证配置ResourceSecurityConfigurer

```

...

@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}

@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey("test123");
    return converter;
}

...

```

修改认证配置，采用JWT方式，设置签名为test123，这里要和认证服务里面的签名保持一致，否则不能正常解析JWT信息。

- 增加获取JWT扩展信息的接口

StockUserController:

```
/**
 * 获取用户JWT扩展信息
 * @return
 */
@RequestMapping("/getJwtInfo")
public ApiRespResult getUserEnhancer() {

    ApiRespResult result = null;
    try {
        // 获取用户JWT扩展信息
        Map<String, Object> userAdditionalInfos = getUserAdditionalInfos();
        result = ApiRespResult.success(userAdditionalInfos);
    } catch (ComponentException e) {
        log.error(e.getMessage(), e);
        result = ApiRespResult.error(e.getErrorCodeEnum());
    } catch (Exception e) {
        log.error(e.getMessage(), e);
        result = ApiRespResult.sysError(e.getMessage());
    }

    return result;
}
```

- 自定义解析JWT数据

增加依赖:

```
<!-- JWT TOKEN 组件 -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.0</version>
</dependency>
```

解析方法:

```

/**
 * 解析获取JWT Token 信息
 * @return
 */
protected String getJwtToken() {
    // 1. 获取Request对象
    HttpServletRequest request = ((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getRequest();
    // 2. 获取token信息
    String token = request.getHeader("Authorization");
    if(null != token) {
        token = token.replace(OAuth2AccessToken.BEARER_TYPE, "").trim();
    }
    return Jwts.parser()
        .setSigningKey("test123".getBytes(StandardCharsets.UTF_8))
        .parseClaimsJws(token)
        .getBody().toString();
}

```

3. 测试验证

- 申请Token

```

POST http://127.0.0.1:10680/oauth/token?
grant_type=password&username=admin&password=admin&scope=server
Accept: */*
Cache-Control: no-cache
Authorization: Basic YXBwOmFwcA==

```

返回Token信息:

```

{
  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpb19pbmZvIjoiaGVrdW4xQG10Y2FzdC5jbXNudWxsIiwidXNlcl9pZCI6MSwidXNlcl9uYW11IjoieWRtaW4iLCJzY29wZSI6WyJzZXJ2ZXIiXSwiYXhwIjojNTk0ODQ5NTg1LCJqdGkiOiI2OWI4MWQzMj05MTk2LTQ5YmItOTU3ZC05YmRlZDM2OTY3ZTAiLCJjbGllbnRfaWQiOiJhcHAifQ.bFBKhpF0IYnJ9dpZG4aPiImlpLECYwK-jTYTPHD2fc_M",
  "token_type": "bearer",
  "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpb19pbmZvIjoiaGVrdW4xQG10Y2FzdC5jbXNudWxsIiwidXNlcl9pZCI6MSwidXNlcl9uYW11IjoieWRtaW4iLCJzY29wZSI6WyJzZXJ2ZXIiXSwiYXhwIjojNTk0ODQ5NTg1LCJqdGkiOiI2OWI4MWQzMj05MTk2LTQ5YmItOTU3ZC05YmRlZDM2OTY3ZTAiLCJjbGllbnRfaWQiOiJhcHAifQ.yHD0U1WtOH_SAGev3mPwD1L1_XucWvtRpTT-upHNqTM",
  "expires_in": 43199,
  "scope": "server",
  "login_info": "hekun1@itcast.cn|null",
  "user_id": 1,
  "user_name": "admin",
  "jti": "69b81d32-9196-49bb-957d-9bded36967e0"
}

```

- 获取JWT扩展用户信息

```
GET 127.0.0.1:10680/user/getJwtInfo
Accept: */*
Cache-Control: no-cache
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpb19pbmZvIjoiaGVrdW4xQG10Y2FzdC5jbXNudWxsIiwidXNlc19pZCI6MSwidXNlc19uYW11IjoiYWRTaW4iLCJzY29wZSI6WyJzZXJ2ZXIiXSwiZXhwIjoxNTk0ODQ5NTg1LCJqdGkiOiI2OWI4MWQzMj05MTk2LTQ5YmItOTU3ZC05YmRlZDM2OTY3ZTAiLCJjbGllbnRfawQiOiJhcHAifQ.bFBKhPf0IYnJ9dpZG4aPIImpLECYwK-jTYTPHd2fc_M
```

返回JWT扩展用户信息:

```
{
  "code": "SYS_200",
  "msg": "成功",
  "extendData": null,
  "data": {
    "login_info": "hekun1@itcast.cn|null",
    "user_id": 1,
    "user_name": "admin",
    "jti": "69b81d32-9196-49bb-957d-9bded36967e0"
  },
  "success": true
}
```