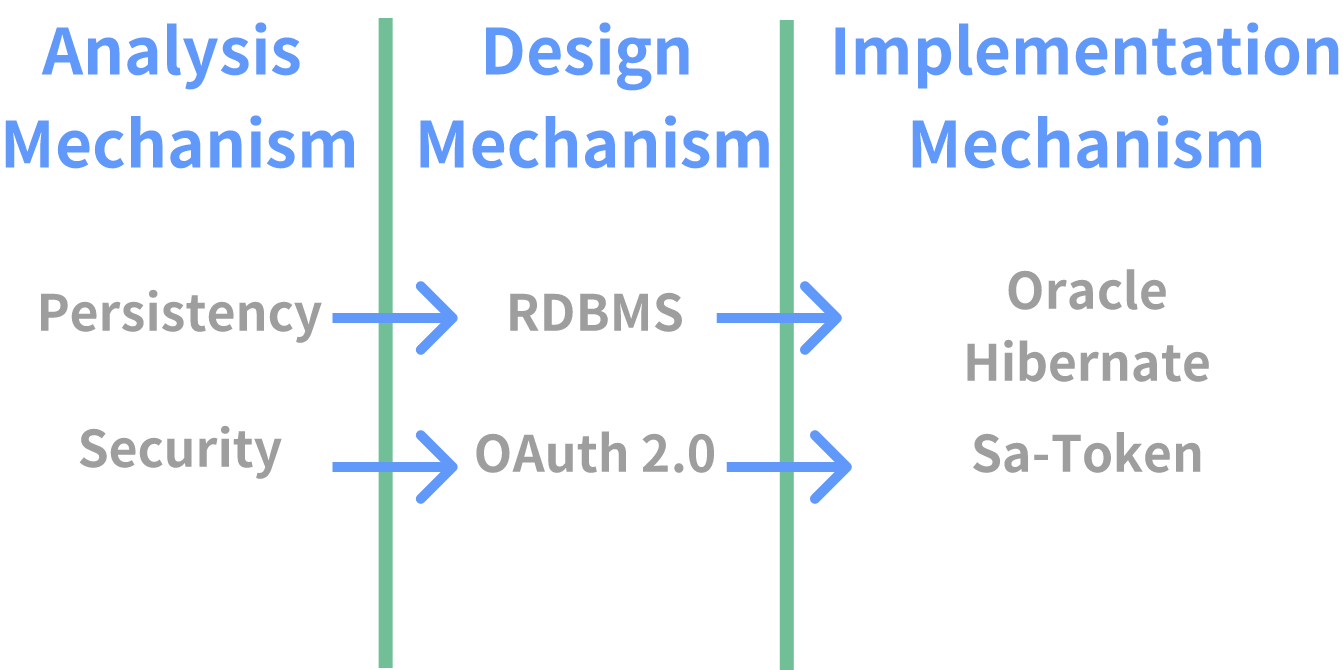
**3 设计机制**

HW的架构设计如下图所示：



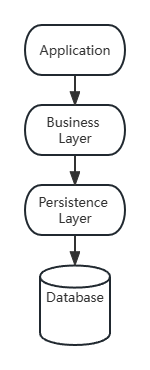
以下将讨论设计机制中的几个案例：

**数据持久存储机制**

数据持久存储机制是指将数据在计算机系统中长期存储的方法和技术。它的存在使得程序可以在多次运行之间保留数据状态，从而提高了程序的可靠性和效率。其中，数据的存储需要使用内存、硬盘等物理介质。但是，内存中的数据会随着计算机的关闭而消失，而磁盘上的数据可以长期保存。因此，数据持久化机制可以确保数据在程序关闭之后得到长期保留。再者，如果数据只存在于内存中，那么一旦程序崩溃或电源故障发生，所有未保存的数据都将丢失。通过使用数据持久化机制，可以将数据写入到硬盘上，从而确保即使系统故障或突然关闭时，也可以恢复数据。同时，数据持久化机制还允许多个用户访问相同的数据，这种情况在共享应用程序或数据库时非常常见。如果数据仅保存在一个用户的内存中，则其他用户无法访问该数据。有了数据持久化，多个用户可以同时访问存储在磁盘上的数据。正如其他应用软件一样，我们的HW中有许多数据需要进行永久保存，从而实现系统的相关功能。例如保存用户的账户信息、留言板内容、随笔内容甚至更重要的是情绪识别对某一特定用户的识别结果。根据我们的要求，我们需要将系统数据存储入关系型数据库中。

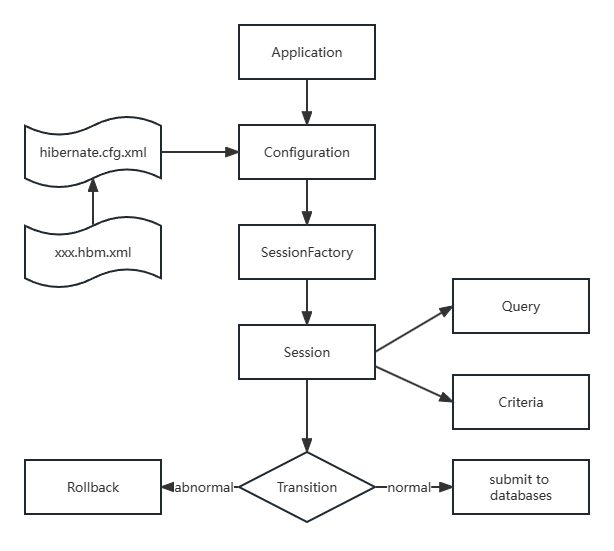
此时，我们需要使用ORM将数据库中的关系数据和面向对象编程语言中的对象进行映射。原因在于，在实际编程中，我们将使用面型对象编程语言，从而表达各种实体之间的继承、关联等关系，而不是使用低级别的SQL代码手足无措；同时，ORM提供了一个清晰的抽象层，隐藏了底层的数据库实现细节，使得应用程序更易于维护和升级；再者，ORM使应用程序更可靠，因为它可以帮助开发人员避免常见的错误，如SQL注入攻击等；最后，ORM框架通常会自动优化生成的SQL查询，从而使应用程序更快速响应，并减少数据库访问次数。

为了实现这个目标，我们在业务层和数据库之间实现了一个持久化层。持久化层是一种中间件，可以自动将对象存储在数据库中，并从数据库加载数据到内存中。通过这样的抽象层，程序将通过会话间接地访问数据库。如下图所示：



我们将选择使用ORM中的Hibernate框架。原因在于Hibernate是一个非常流行的开源ORM框架，它提供了一种将Java对象映射到数据库表的方式。通过使用Hibernate，我们可以通过简单的注释或XML配置来描述数据模型和映射关系，Hibernate会自动处理对象与关系数据之间的转换。同时，Hibernate提供了面向对象的查询语言HQL，允许使用面向对象的方式进行数据库查询。同时，Hibernate还支持本地SQL查询，使得程序员可以在需要时直接编写原生SQL语句。不仅如此，Hibernate支持多种主流的关系型数据库，包括MySQL、Oracle、Microsoft SQL Server等，并且也支持NoSQL数据库MongoDB。最后，除了自动化ORM操作外，Hibernate还提供了缓存机制，可以有效地减少数据库访问的次数，以提高应用程序的性能。Hibernate还提供了事务管理、连接池管理等功能，使得开发人员可以更方便地编写数据库相关的代码。

Hibernate机制如下：



配置(Configuration)对象通常是Hibernate应用程序中创建的第一个对象。它通常只在应用程序初始化期间创建一次，表示Hibernate所需的配置或属性文件。

Configuration对象负责管理Hibernate运行所需的配置信息，并读取和处理配置文件（例如hibernate.cfg.xml）中指定的配置属性。使用这些信息，它创建SessionFactory类的实例。SessionFactory是任何Hibernate应用程序的关键组件，因为它用于获取Hibernate Sessions的实例，这些实例用于执行数据库操作。因此，Configuration对象在任何Hibernate应用程序中都扮演着至关重要的角色，通过提供必要的配置信息来允许Hibernate连接到数据库，并创建SessionFactory，该SessionFactory用于管理Hibernate Sessions的生命周期。

会话工厂(SessionFactory)是Hibernate应用程序中最为关键的对象之一，它负责管理Hibernate Session的创建和生命周期，并处理所有与数据库交互的操作。在Hibernate中，Session是执行数据库操作的主要接口。同时，SessionFactory是一个线程安全的对象，通常只在应用程序启动时创建一次，然后在整个应用程序运行期间被重复使用。这种设计模式称为单例模式(Singleton Pattern)。再者，SessionFactory的创建是一个开销比较大的过程，因为它需要读取配置文件、扫描实体类等操作。但是，在SessionFactory实例化完成之后，它会缓存各种映射信息和预编译的HQL语句，从而提高性能。SessionFactory可以通过Configuration对象来创建，也可以通过ServiceRegistry构建器来创建。创建SessionFactory时，需要传递一个Configuration对象或者一个包含了Hibernate配置信息的Properties对象。SessionFactory则会根据这些配置信息来生成一个或多个与数据库的连接池，并且管理这些连接池的打开和关闭。最后SessionFactory创建好之后，我们就可以使用它来获取Session了。在Hibernate中，Session是一个轻量级的、非线程安全的对象，它代表了持久化上下文(Persistence Context)，并提供了对数据库的CRUD操作。每个Session实例都属于某个SessionFactory，它使用SessionFactory中提供的连接池进行数据库操作。总之，SessionFactory是Hibernate应用程序的关键组件之一，它负责管理Hibernate Session的创建和生命周期，并提供了对数据库的连接池、缓存机制等进行管理的功能。对于需要频繁进行数据库操作的应用程序来说，SessionFactory的优化和使用非常重要。

在Hibernate中，Session是一个重要的概念，它代表了应用程序与数据库之间的一次会话。通过Session对象，我们可以执行各种数据库操作，比如插入、更新、删除和查询数据等。同时，Session也负责管理持久化对象的状态，确保与数据库的同步。下面是Session的一些主要功能：

·数据库连接管理：Session封装了JDBC的连接，并负责管理数据库连接的创建、打开、关闭和回收。这样，我们就无需手动处理底层的数据库连接，而是可以专注于业务逻辑的实现。

·对象状态管理：Hibernate使用对象关系映射（ORM）技术，将Java对象映射到数据库表中的记录。Session负责跟踪持久化对象的状态，即临时状态、持久状态和游离状态。在不同状态之间转换时，Session会自动维护对象与数据库记录之间的同步。

·数据库操作：Session提供一系列API，可以执行各种数据库操作。比如，我们可以使用Session的save()和persist()方法来插入新的记录；使用update()和merge()方法来更新已有记录；使用delete()方法删除记录；使用get()和load()方法查询记录等。

·事务管理：Session还提供了事务管理的功能。我们可以使用Session的beginTransaction()方法开启事务，然后在事务中执行多个数据库操作。如果所有操作都成功，则提交事务；否则回滚事务。

·查询功能：除了直接操作数据库外，我们还可以使用Session进行查询操作。Hibernate提供了多种查询方式，比如HQL、Criteria和Native SQL等。我们可以使用这些查询方式来检索数据、进行聚合计算等复杂操作。

Hibernate中的Transaction（事务）用于管理对数据库的多个操作，从而确保这些操作要么全部成功要么全部失败回滚。在Hibernate中，Transaction对象是通过Session对象创建的。一个Transaction可以通过commit()方法来提交，也可以通过rollback()方法来回滚。如果不进行提交或回滚，则Transaction将处于挂起状态，直到Session关闭。

Query是一个接口，用于执行HQL和SQL查询，并返回结果集。它提供了许多方法来设置查询参数、限制结果集大小等功能。在使用Query之前，必须先获得一个Session对象。然后可以通过调用Session的createQuery()方法创建一个Query对象。一旦创建了Query对象，就可以使用setFirstResult()和setMaxResults()方法来限制结果集的大小。setFirstResult()方法指定要从结果集中返回的第一个记录的索引，而setMaxResults()方法指定最大返回记录数。

在Hibernate中，Criteria是一种用于创建和执行类型安全查询的API。它通过构建一个包含各种查询条件的条件对象来实现这一点，并将它们应用于Hibernate映射类。使用Criteria API可以避免手写SQL或HQL语句，还可以提供更好的可读性和易于维护性，因为它们使用类型安全的方法和属性而不是字符串。

**权限管理**

为了确保每个用户角色都能顺畅地使用系统中的服务和资源，我们需要实现一个全面、高效且易于使用的用户访问控制机制。这个机制应该能够根据不同角色的权限限制，确保他们只能访问到他们被授权的内容，并且能够快速地执行访问控制操作，以提高系统的安全性和可靠性。

我们采用微服务架构并结合常见的权限管理实践，以提供更加细粒度、可靠的用户访问控制机制。这种架构模式具有很多优点。一方面，每个微服务可以通过中心化的授权系统进行权限验证，同时也可以在自己的内部设计更加灵活的权限控制机制。另一方面，在可靠性方面，相比于传统的单一应用模式，基于微服务的系统可以更快地提供服务，并且当某个微服务出现故障时，其他微服务不会受到影响。此外，每个微服务都可以充分利用其硬件资源，而独立扩展某个微服务也不会对其他微服务造成影响。综上所述，微服务架构与权限管理实践的相结合可以有效提高系统的安全性和可靠性，为用户提供更好的使用体验。

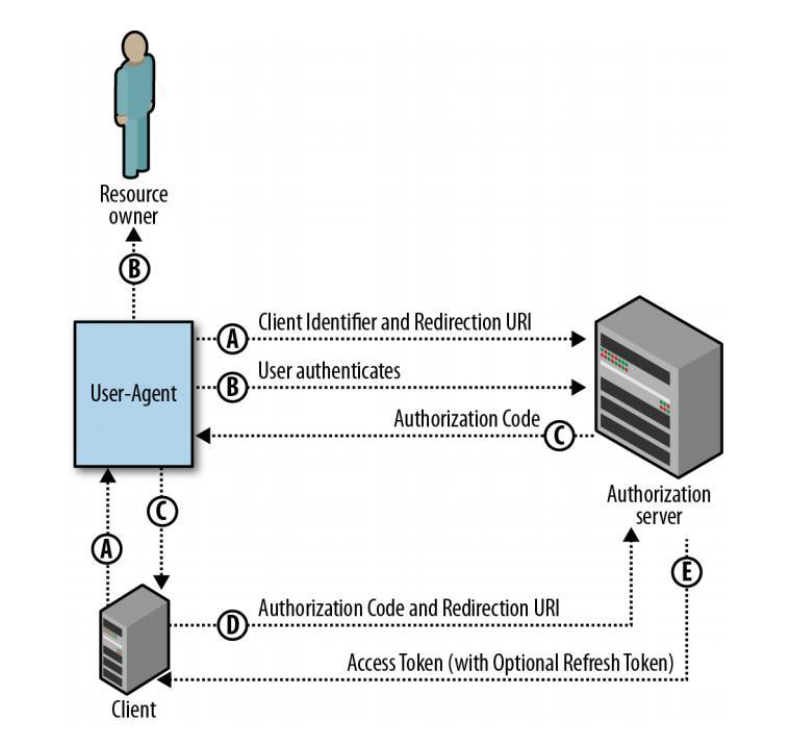
在我们的项目中，我们采用 OAuth 2.0 标准协议来进行授权管理。OAuth 2.0 是一种基于 Token 的授权管理系统，相对于基于 cookie 的认证方式，它具有更多优点。例如，基于 Token 的认证机制是无状态的，可以有效防止跨站请求伪造 (CSRF) 攻击，并且支持多站点使用。因此，我们选择使用 OAuth 2.0 协议来进行权限认证，以提高我们系统的安全性和可靠性。

为了在微服务架构中使用 OAuth 2.0，我们采用了常规的 API 网关组件。API 网关是微服务架构的标准组件之一，负责接收来自客户端的请求，并将它们路由到相应的微服务。在这个过程中，API 网关还可以对请求进行鉴权和授权操作，以确保只有经过授权的用户才能访问相应的服务。

具体来说，我们的系统通过引入一个授权服务作为 API 网关的一部分，实现了统一的权限管理机制。当客户端发起请求时，API 网关会将请求发送给授权服务进行身份验证，并获取相应的访问令牌(Token)。然后，API 网关会将访问令牌附加在请求头中，将请求路由到相应的微服务。在微服务内部，可以通过读取请求头中的访问令牌进行权限校验，以确保只有授权用户才能访问对应的服务。这种方式不仅可以简化权限管理的流程，还可以提高系统运行效率，减少客户端与后端的往返次数。因此，通过引入授权服务和 API 网关的组合方式，我们的系统可以高效、安全地管理用户的权限，并且更好地适应微服务架构。

在我们的实际开发中，我们选择了 Sa-Token 作为授权服务的框架。相比于其他框架，Sa-Token 具有轻量级的特性，使得它在微服务架构中具有更好的灵活性和适应性，便于我们将其整合到整个系统中。同时，Sa-Token 还提供了 Sa-OAuth2 模块，基于 RFC-6749 标准编写，可以非常方便地实现 OAuth 2.0 授权认证。除此之外，Sa-Token 还支持常见的二级认证，这在我们的项目中也是较为常见的需求。

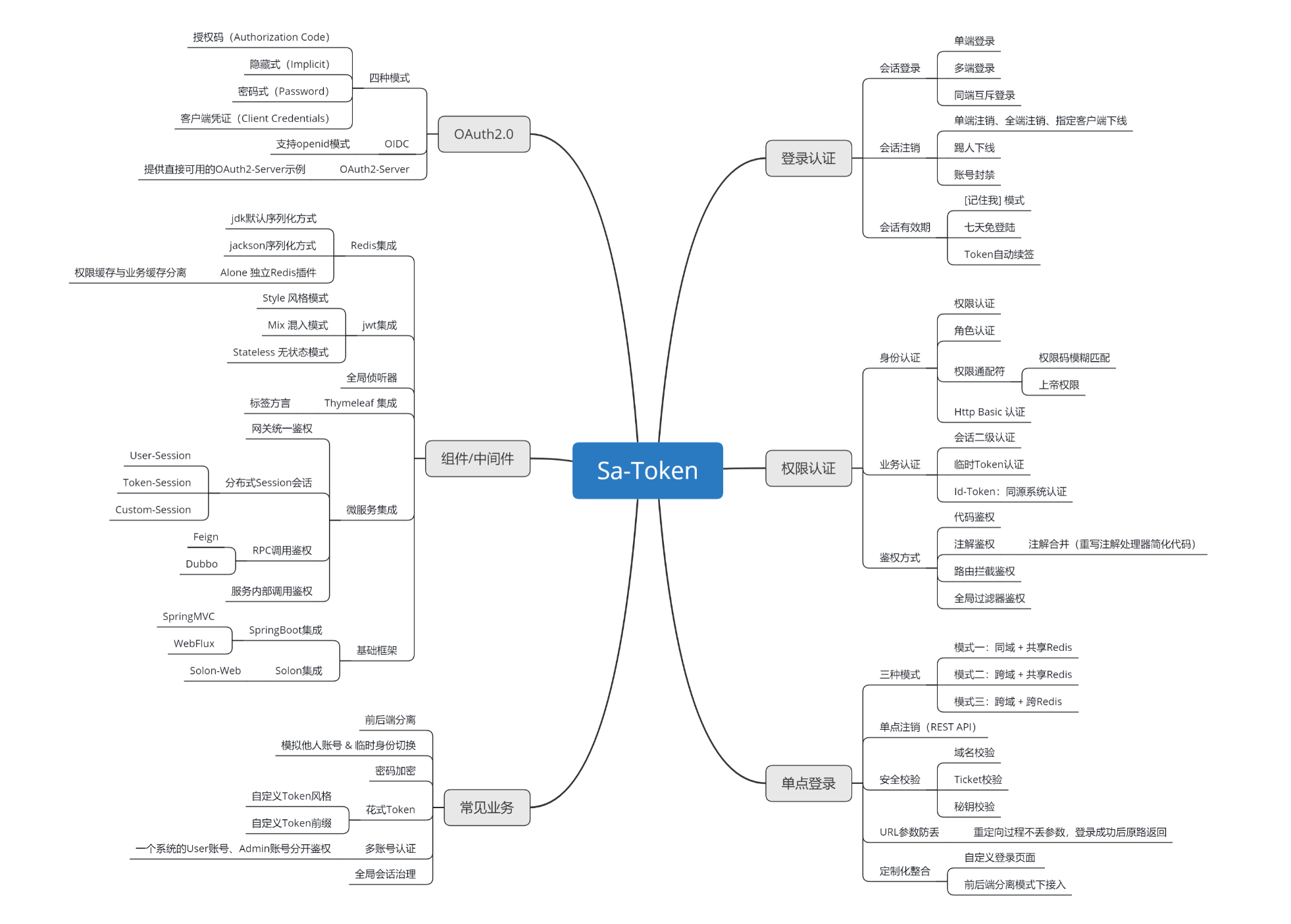
OAuth 2.0授权过程如下图所示：



典型的 Sa-Token 认证流程如下：首先，客户端向 API 网关发送请求，并附带访问令牌(Token)。API 网关收到请求后，会将 Token 发送给 Sa-Token 进行验证。Sa-Token 会对 Token 进行解析，并根据解析结果判断请求是否有效。如果 Token 验证通过，则 API 网关会将请求路由到相应的微服务中进行处理；否则，Sa-Token 将拒绝该请求并返回错误信息。

在 Sa-Token 中，Token 的生成、解析和管理等操作都由框架自动完成，无需手动编写代码。同时，Sa-Token 还提供了丰富的权限控制功能，例如角色、权限、登录设备等，可以满足复杂的权限管理需求。总之，我们选择 Sa-Token 作为授权服务框架，可以更加高效、安全地管理用户的权限，并提供灵活的二级认证机制，为我们的系统开发带来了很大的便利。

Sa-Token功能如下所示：



Sa-Token流程如下图所示：

