1. 微信支付

Native支付是指商户系统按微信支付协议生成支付二维码，用户再用微信"扫一扫"完成支付的模式。

1.1 公司的个体工商户

公司和个体工商户是两种不同的商业实体类型，它们在法律地位、组织结构、税务处理等方面有着显著的区别。了解这些区别有助于在创业或经营活动中做出更合适的选择。下面是两者之间的主要区别：

法律地位

1. **公司**:
    - 公司是一种具有独立法人资格的商业实体。
    - 它可以拥有财产、签订合同、承担法律责任，并且与其股东（所有者）的法律身份是分开的。
    - 公司的责任一般限于其拥有的资产。
2. **个体工商户**:
    - 个体工商户不具有独立的法人资格，它是其所有者的延伸。
    - 个体工商户的所有者对企业的债务和责任承担个人无限责任。
    - 个体工商户通常适用于小规模的家庭式经营或个人业务。

组织结构

1. **公司**:
    - 公司的结构相对复杂，可能包括股东、董事会、管理层等。
    - 公司的经营决策通常由董事会或管理层做出，而股东则拥有投票权和获得利润的权利。
2. **个体工商户**:
    - 个体工商户的结构简单，所有的经营决策由业主个人做出。
    - 通常没有正式的管理层或员工结构。

税务处理

1. **公司**:
    - 公司需按照公司税率纳税，通常需要缴纳企业所得税。
    - 公司的财务和税务报告要求更为严格。
2. **个体工商户**:
    - 个体工商户的税务通常更简单，税率可能低于公司

- 纳税申报通常基于个人所得，合并在个人所得税中处理。

融资能力

1. **公司**:
   - 公司通常具有更强的融资能力，可以通过发行股票或债券等方式筹集资金。
   - 由于法人身份和结构，公司能够更容易获得银行贷款和投资。
2. **个体工商户**:
   - 个体工商户的融资渠道相对有限，通常依赖于个人资金或小额贷款。
   - 筹资能力受限于个人信用和资源。

1.2 超级个体

“超级个体”（Super Individual）是一个相对较新的概念，通常指在某一领域或多个领域拥有高度专业技能和知识的个人，他们利用技术和网络资源，独立或通过灵活的工作安排，在全球范围内提供服务或影响力。这个概念在数字化和远程工作日益普及的背景下变得越来越重要。

以下是“超级个体”的一些主要特征和背景：

特征

1. **高度专业化的技能和知识**:
   - 超级个体通常在他们选择的领域内具有深厚的专业知识和技能，例如编程、设计、营销、咨询等。
2. **利用数字工具和平台**:
   - 他们充分利用数字技术、在线平台和社交媒体等工具来扩大他们的工作范围和影响力。
3. **独立和自主**:
   - 超级个体往往是自由职业者、独立承包商或企业家，他们自主管理工作，不受传统雇佣关系的限制。
4. **全球范围的影响力和服务**:
   - 他们的服务和影响力通常不局限于本地市场，而是面向全球客户或受众。
5. **网络协作和众包**:
   - 虽然独立工作，但超级个体经常通过网络与其他专业人士合作，参与更大的项目或共享资源。

背景和影响

- **数字化转型**:
  - 信息技术和互联网的发展使个人能够更容易地访问全球市场，使用先进的工具和平台来提供服务。
- **远程工作文化**:
  - 远程工作的普及为个体提供了更多灵活性，使他们能够在任何地方工作，并与全球客户建立联系。
- **经济和工作市场的变化**:
  - 经济的全球化和工作市场的变化促使越来越多的人选择自由职业和独立承包的工作方式，而不是传统的全职工作。

结论

超级个体代表了一种新兴的工作和经济模式，他们的兴起反映了当代社会和技术环境的变化。随着技术的进步和工作方式的演变，预计这一群体将继续增长，并对全球经济和创新产生重要影响。

2. 支付接入前准备

Native支付接入前准备

2.1 申请APPID

要在微信支付中申请APPID，您需要按照以下步骤进行操作：

1. **注册微信公众平台账号**：首先，您需要访问微信公众平台的官方网站并注册一个账号。这通常需要您提供一些基本信息，如邮箱地址和手机号码。
2. **选择账号类型**：微信公众平台提供不同类型的账号，例如服务号、订阅号等。如果您的主要目的是使用微信支付，那么通常建议注册服务号，因为它提供更全面的功能，包括微信支付。
3. **验证企业信息**：注册服务号后，您需要验证企业信息。这通常涉及提供您的企业注册信息、组织机构代码证、税务登记证等相关文件。这一步是确保账号安全和合规性的关键。
4. **申请微信支付功能**：在您的微信公众账号后台，有一个微信支付的选项。您需要点击申请，然后按照指示完成相关的步骤。这可能包括提供更多的企业信息、银行账户信息以及签署相关的协议。
5. **等待审核**：提交申请后，微信团队将对您的资料进行审核。这个过程可能需要几天到几周的时间。
6. **获取APPID**：一旦您的微信支付功能被激活，您的微信公众账号就会有一个与之关联的APPID。您可以在微信公众平台后台找到这个APPID。
7. **开发者配置**：如果您需要将微信支付集成到您的网站或应用中，您可能还需要进行一些开发者配置，包括设置API密钥等。

2.2 申请商户号

在微信支付中申请商户号（MCH ID）是一个重要步骤，用于使企业能够接收和处理微信支付交易。以下是申请MCH ID的基本步骤：

1. **注册微信商户平台账号**：
   - 访问微信支付的官方网站。
   - 在微信支付官网上选择“立即注册”，进行注册流程。
2. **选择账号类型**：
   - 根据您的业务需求选择合适的账号类型，如普通商户、服务商等。
   - 提供相关的企业信息，如营业执照、组织机构代码证等。
3. **填写商户信息**：
   - 完整填写商户的详细信息，包括企业信息、经营信息、结算信息等。
   - 这些信息需要准确无误，因为它们将用于审核和后续的交易处理。
4. **上传相关证件和资料**：
   - 根据提示上传所需的企业相关证件和资料。
   - 这可能包括营业执照副本、法人身份证、银行账户信息等。
5. **提交审核**：
   - 审核所有信息无误后，提交申请以进行审核。
   - 微信支付团队将对提交的信息和文件进行审核。
6. **审核过程**：

- 审核过程可能需要几天到几周的时间。
- 在此期间，保持关注您的注册邮箱或微信支付账户，以便了解审核进度或补充信息。

7. **获取MCH ID**：
   - 一旦审核通过，您将收到一个商户号（MCH ID）。
   - 这个MCH ID是您在微信支付系统中的唯一标识，用于处理和管理所有的微信支付交易。
8. **配置支付功能**：
   - 根据业务需求配置您的支付接口和API。
   - 可以设置API密钥，并确保所有的配置都符合安全和合规性要求。

2.3 给商户号指定APPID

在微信支付中，给商户号指定APPID是一个关键步骤，它允许将您的微信支付商户账户（MCH ID）与特定的微信公众号或小程序（APPID）关联。以下是如何进行这一操作的步骤：

1. **登录微信支付商户平台**：
   - 首先，您需要登录到微信支付商户平台（微信商户平台）。
   - 使用您的商户账号和密码登录。
2. **访问开发配置**：
   - 在商户平台的主界面上，寻找“开发配置”或类似的选项。
   - 这一选项通常在“产品中心”、“账户设置”或“我的商户”等标签下。
3. **配置API安全**：
   - 在开发配置中，可能需要先设置API安全项，包括API证书和API密钥。
   - 这些设置对于确保交易的安全性非常重要。
4. **输入APPID**：
   - 在相同的配置页面中，找到指定APPID的部分。
   - 输入您想要关联的微信公众号或小程序的APPID。
5. **保存设置**：
   - 完成APPID的输入后，保存您的设置。
   - 确认所有信息准确无误，尤其是APPID的正确性。
6. **等待审核（如果需要的话）**：
   - 某些情况下，微信支付可能需要审核您的配置。
   - 在审核过程中，请保持关注，以便及时响应可能的查询或要求。
7. **测试配置**：
   - 一旦配置完成并且通过审核（如果有的话），建议进行一次测试交易，以确保支付功能正常运作。
   - 您可以在微信公众号或小程序中设置一个小额支付，来验证配置的有效性。

2.4 APPID指定商户号

在微信支付中，将特定的微信支付商户号（MCH ID）与您的微信公众号或小程序（APPID）关联，是一个重要的设置步骤，使您的应用能够处理微信支付交易。以下是如何进行这一操作的步骤：

1. **登录微信公众平台或小程序管理后台**：

- 首先，登录到您的微信公众号或小程序的管理后台。
- 网址通常为：微信公众平台 或 微信小程序管理后台。

2. **访问支付配置**：
   - 在管理后台中，找到“设置”或“开发设置”部分。
   - 在该部分中寻找与支付相关的配置选项。
3. **输入商户号（MCH ID）**：
   - 在支付配置页面，您将找到一个用于输入商户号（MCH ID）的字段。
   - 输入您的微信支付商户号。确保输入无误，因为这是您接收支付的关键标识。
4. **配置API密钥**：
   - 在同一支付配置中，您可能还需要设置API密钥。
   - API密钥是从微信支付商户平台生成的，用于确保支付过程的安全性。
5. **保存配置**：
   - 输入商户号和设置好API密钥后，保存您的配置。
   - 确保所有信息准确，以避免交易处理问题。
6. **测试支付功能**：
   - 配置完成后，进行一次测试交易以确保一切正常。
   - 您可以在微信公众号或小程序中设置一个小额的测试交易。
7. **审核和激活**：
   - 某些情况下，微信可能需要对您的支付配置进行审核。
   - 完成审核后，您的微信公众号或小程序即可使用绑定的商户号处理支付。

2.4 配置APIV3密钥

在微信支付中配置APIv3密钥是一个重要的步骤，它用于加强API调用的安全性。APIv3密钥是微信支付API接口调用的一个安全机制，用于生成签名和验证应答，确保数据传输的安全。以下是配置APIv3密钥的基本步骤：

1. **登录微信支付商户平台**：
   - 首先，访问微信支付商户平台并使用您的商户号登录。
2. **访问API安全设置**：
   - 在商户平台主界面，找到“API安全”相关的部分。
   - 这一部分通常在“账户设置”或“安全中心”下。
3. **生成APIv3密钥**：
   - 在API安全设置中，寻找“设置APIv3密钥”的选项。
   - 点击生成密钥，系统将提示您确认生成新的APIv3密钥。
   - 请注意，一旦生成新密钥，之前的APIv3密钥将不再有效。
4. **记录APIv3密钥**：
   - 在生成密钥后，系统会显示新的APIv3密钥。
   - 请立即复制并妥善保管这个密钥。出于安全考虑，这个密钥只显示一次，微信支付商户平台不会存储。
5. **更新您的系统**：

- 使用新的APIv3密钥更新您的支付系统或应用程序中的配置。
- 确保在所有需要使用微信支付API的地方更新此密钥。

6. **测试确保配置正确**：
   - 在更换密钥后，进行一些测试交易以确保一切正常工作。
   - 确认新密钥可以成功用于支付请求和处理回调。
7. **保持密钥安全**：
   - APIv3密钥非常重要，需要严格保密。
   - 避免在不安全的地方存储或传输密钥，确保只有授权人员才能访问。

2.5 申请并下载商户证书

申请微信支付商户证书是为了确保您的交易安全和数据加密。商户证书主要用于签名和验证微信支付的API调用。以下是申请微信支付商户证书的基本步骤：

1. **登录微信支付商户平台**：
   - 首先，访问微信支付商户平台官网 微信支付商户平台。
   - 使用您的商户号和密码登录到平台。
2. **访问API安全设置**：
   - 登录后，在平台主界面找到“API安全”或相关的选项。
   - 这一选项通常位于“账户设置”、“安全中心”或“产品中心”的子菜单中。
3. **申请商户证书**：
   - 在API安全的设置页面中，寻找“API证书”或“申请证书”的选项。
   - 选择申请API证书。过程中，系统可能会要求您提供一些必要的信息或文件，例如公司信息、管理员信息等。
4. **验证信息**：
   - 根据系统提示，完成验证步骤。这可能包括短信验证、邮件验证等。
   - 确保所提供的信息准确无误，以便顺利完成验证。
5. **下载证书**：
   - 审核通过后，您将能够下载API证书。
   - API证书通常包括一个证书文件（`.pem`格式）和一个密钥文件（`.p12`格式）。
6. **安装和配置证书**：
   - 将下载的证书安装到您的服务器或支付系统中。
   - 在您的支付应用程序中配置证书路径和相关参数。
7. **测试证书**：
   - 配置完成后，进行测试以确保证书正确安装并且可以用于微信支付API的调用。
   - 测试可以包括发起一个支付请求或处理一个支付通知。
8. **保持证书安全**：
   - 保护好您的商户证书，不要泄露给未经授权的人员。
   - 定期检查和更新证书，以确保支付安全。

3. 加密与安全

3.1 概念

1. **明文**：
   - 明文是指原始信息或数据的原始形式，未经加密的状态，可以是文本、数字或任何其他形式的数据。
   - 明文是容易读懂的，没有经过任何形式的隐藏或加密处理。
2. **密文**：
   - 密文是指经过加密后的信息或数据。密文通常看起来是随机的或没有明显意义的字符序列。
   - 密文的目的是为了保护数据的隐私性，防止未经授权的人员理解或篡改数据。
3. **密钥**：
   - 密钥是用于加密和解密数据的工具。它可以是一个数字、一串字符或更复杂的数据结构。
   - 在加密过程中，密钥用于转换明文为密文。在解密过程中，又用于将密文转换回明文。
   - 密钥的安全性对保护数据至关重要。
4. **加密**：
   - 加密是一个过程，通过它，明文使用一定的方法（加密算法）和密钥被转换成密文。
   - 加密的目的是保护数据的机密性，使未授权的人无法理解信息的真实内容。
5. **解密**：
   - 解密是加密的逆过程。它使用密钥将密文转换回原始的明文形式。
   - 只有拥有正确密钥的人才能解密并获得原始数据。
6. **加密算法**：
   - 加密算法是一系列用于加密和解密数据的指令或规则。
   - 有多种加密算法，每种都有其独特的特点和用途，如AES、RSA、DES等。
   - 不同的算法可能使用不同类型的密钥（如对称密钥或非对称密钥）。

3.2 对称加密和非对称加密

对称加密和非对称加密是两种主要的加密方法，它们在保护数据安全性方面发挥着关键作用。下面将分别解释这两种加密方法的特点和工作原理。

3.2.1 对称加密

对称加密是一种加密方法，其中加密和解密使用相同的密钥。其特点和工作原理如下：

- **密钥共享**：发送方和接收方使用相同的密钥进行加密和解密。
- **加密速度快**：由于算法相对简单，对称加密通常比非对称加密更快，适合大量数据的加密。
- **主要挑战**：密钥的分发和管理。因为密钥必须在通信双方之间安全共享，任何泄露都可能导致安全风险。
- **常见算法**：AES（高级加密标准）、DES（数据加密标准）、3DES（三重数据加密算法）等。

对称加密适用于那些对数据处理速度要求较高且能够安全管理密钥的场景。

3.2.2 非对称加密

非对称加密，也称为公钥加密，使用一对密钥：一个公钥和一个私钥。这两个密钥在数学上是相关联的，但不能通过其中一个推导出另一个。其特点和工作原理如下：

- **公钥和私钥**：公钥可以公开分享，用于加密数据；私钥是保密的，只有所有者才知道，用于解密数据。
- **安全性高**：由于密钥对的独特性，非对称加密通常被认为比对称加密更安全。
- **速度较慢**：相比对称加密，非对称加密在处理数据时速度较慢，因此不太适合大量数据的加密。
- **常见算法**：RSA、ECC（椭圆曲线加密）、Diffie-Hellman密钥交换等。

非对称加密主要用于安全密钥交换、数字签名和身份验证等场景。

3.2.2.1 加密信息

非对称加密用于加密信息时，主要目的是保证信息传输的机密性。其过程如下：

1. **使用公钥加密**：
   - 发送方使用接收方的公钥对信息进行加密。
   - 由于只有接收方拥有对应的私钥，因此只有接收方能解密这个信息。
2. **使用私钥解密**：
   - 接收方使用自己的私钥解密信息。
   - 这确保了即使加密的信息在传输过程中被拦截，拦截者也无法解读信息内容。
3. **保护机密性**：
   - 这种方法主要用于保护信息内容不被未授权的第三方读取。

3.2.2.2 身份认证

在身份认证中，非对称加密主要用于验证一个实体的身份，确保消息的来源是可信的。其过程通常涉及数字签名，如下所示：

1. **使用私钥生成签名**：
   - 发送方使用自己的私钥对信息或信息的散列（hash）值进行加密，生成数字签名。
   - 这个签名绑定于特定的信息和发送者的身份。
2. **使用公钥验证签名**：
   - 接收方使用发送方的公钥来解密签名。
   - 接收方同时对原始信息进行同样的散列处理，然后比较散列值是否一致。
3. **确保身份和完整性**：
   - 如果使用公钥解密的结果与原始信息的散列值一致，说明信息确实来自声明的发送方，并且在传输过程中未被篡改。
   - 这种方法主要用于验证信息的真实性和发送者的身份。

3.2.2.3 综合应用

在实际应用中，非对称加密经常被用于同时实现加密信息和身份认证。例如，在发送一个加密电子邮件时，发送方可以使用接收方的公钥对信息进行加密（保护机密性），同时使用自己的私钥生成数字签名（保证身份认证）。接收方则使用自己的私钥解密信息，并使用发送方的公钥验证数字签名。

这种综合应用提供了一个既安全又有效的方式来保护信息内容，同时验证通信双方的身份，是现代数字通信和网络安全的重要组成部分。

3.2.3 结合使用

在实际应用中，对称加密和非对称加密往往结合使用。例如，非对称加密可以用于安全地交换对称加密的密钥，一旦密钥交换完成，数据传输则使用对称加密进行，这样既保证了密钥交换的安全性，又能利用对称加密处理数据的高效性。这种方法在许多现代加密协议和通信标准中被广泛采用，如HTTPS、SSL/TLS等。

3.3 摘要算法

摘要算法，也被称为散列函数（Hash Function），是一种在计算机科学和密码学中广泛使用的重要技术。它的主要作用是将任意长度的输入数据（通常称为“消息”）转换为固定长度的、通常较短的值，这个值被称为散列值或摘要。

3.3.1 散列函数的关键特性

1. **确定性**：对于同一个输入，散列函数总是产生相同的输出。
2. **高效性**：散列函数的计算过程快速，对于任何大小的输入都能在短时间内产生散列值。
3. **不可逆性**：从散列值不可能反推出原始的输入数据。这是因为散列函数是单向的。
4. **冲突抵抗**：理想的散列函数应该最小化冲突，即两个不同的输入产生相同散列值的情况。完全避免冲突是不可能的，因为输出的长度是固定的，但好的散列函数能使这种情况极其罕见。

3.3.2 散列函数的应用

1. **数据完整性验证**：通过比较数据的散列值，可以验证数据是否在传输或存储过程中被篡改。
2. **安全密码存储**：存储用户密码的散列值，而不是密码本身，以提高安全性。
3. **数字签名**：散列函数用于生成数字签名，保证消息的真实性和完整性。

3.3.3 常见的散列函数

- **MD5**：产生128位散列值，但现在已被认为不够安全，容易受到碰撞攻击。
- **SHA-1**：产生160位散列值，安全性比MD5高，但在高安全需求的领域也已逐渐被淘汰。
- **SHA-256**：属于SHA-2家族，产生256位散列值，目前广泛用于加密货币和网络安全领域。

3.4 数字签名

数字签名是一种用于验证数字信息完整性和真实性的技术，广泛应用于电子商务、软件分发、电子邮件等领域。它使用加密算法来模拟传统的手写签名和印章，但在数字环境中提供更高的安全保障。

3.4.1 原理和过程

数字签名通常基于非对称加密技术，涉及一个私钥（用于签名）和一个公钥（用于验证签名）。其基本过程包括：

1. **签名生成**：
    - **消息摘要**：首先，对需要签名的消息（如文档、电子邮件或软件）进行散列处理，生成一个唯一的摘要（哈希值）。
    - **加密摘要**：然后，使用发送者的私钥对这个摘要进行加密，生成数字签名。
2. **签名验证**：
    - **解密摘要**：接收方使用发送方的公钥对数字签名进行解密，得到摘要。

- **比较摘要**：接收方同样对原始消息进行散列处理，生成摘要，并将其与解密得到的摘要进行比较。
- 如果两个摘要相同，则验证成功，表明消息未被篡改，并确实来自声称的发送者。

3.4.2 特点和优势

- **验证身份**：数字签名可以验证消息的发送者身份，确保消息来源的真实性。
- **保证完整性**：通过比较散列值，可以确保消息在传输过程中未被篡改。
- **不可否认性**：发送者不能否认他们签名的消息，因为只有他们的私钥才能生成签名。

3.5 数字证书

数字证书是一种用于验证实体（如个人、服务器、客户端或组织）身份的电子文档。它使用公钥基础设施（PKI）技术，通过将公钥与身份信息相结合，提供了一种安全验证方式。数字证书类似于现实世界中的身份证或护照。

3.5.1 核心组成

数字证书通常包含以下信息：

1. **证书持有者的身份信息**：如名称、电子邮件地址、组织信息等。
2. **证书持有者的公钥**：用于非对称加密和数字签名验证。
3. **颁发机构（CA）的信息**：证书是由谁颁发的，比如VeriSign、Let's Encrypt等。
4. **有效期限**：证书的有效开始和结束日期。
5. **证书序列号**：唯一标识证书的编号。
6. **数字签名**：证书颁发机构（CA）使用自己的私钥对证书进行的签名，以确保证书的真实性。

3.5.2 功能和作用

1. **身份验证**：证书确保了公钥属于声明的实体，帮助用户或系统验证对方的身份。
2. **建立安全通信**：如在HTTPS中，服务器的数字证书用于建立安全的SSL/TLS连接。
3. **数字签名和加密**：证书中的公钥可以用于解密由相应私钥加密的数据，或验证数字签名。

3.5.3 颁发过程

1. **生成密钥对**：证书申请者首先生成一对密钥（公钥和私钥）。
2. **提交证书签名请求（CSR）**：随后，申请者创建CSR，其中包括公钥和身份信息。
3. **CA处理请求**：证书颁发机构验证申请者的身份信息，然后使用CA的私钥对包含申请者公钥的证书进行签名。
4. **颁发证书**：一旦验证完成，CA将签名的数字证书发放给申请者。

3.5.4 应用场景

- **网站安全**：用于HTTPS，加密网站和访问者之间的通信。
- **电子邮件加密和签名**：证书中的公钥用于验证发送方的身份，以及加密和解密电子邮件。
- **代码签名**：开发者使用证书来签名他们的软件，用户可以验证软件的真实性。

3.6 商户和微信证书

- 商户API证书是指由商户申请的，包含商户的商户号、公司名称、公钥信息的证书。
- 什么是商户API证书？如何获取商户API证书？
- 商户申请商户 API 证书时，证书工具会生成商户私钥，并保存在本地证书文件夹的文件 apiclient_key.pem 中。私钥也可通过工具从商户的 p12 证书中导出。请妥善保管好商户私钥文件
- https://pay.weixin.qq.com/docs/merchant/development/interface-rules/privatekey-and-certificate.html是由 微信支付 负责申请和管理的，该证书包含了微信支付平台的身份标识和公钥信息。
- 商户签名需使用商户私钥。请将商户API证书序列号放在请求HTTP头部的 Authorization 字段中的 serial_no 部分。
- 微信支付签名需使用微信支付平台私钥。微信支付平台证书序列号会包含在响应HTTP头部的 Wechatpay-Serial 字段中。
- 如何查看商户API证书序列号？

4.资源

4.1 概念

- APIv3概述
- SDK接入
- 接口基本规则
- 私钥和证书
- API v3密钥
- 签名
- SDK和开发工具
- Postman调试工具
- 平台证书下载工具
- 验签工具

4.2 开发准备

- 申请AppID和mchid
- 配置API key
- 下载并配置商户证书

4.3 算法

- 如何生成请求签名
- 如何验证签名
- 如何解密证书和回调报文
- 如何加解密敏感信息

4.4 Native支付

- 产品介绍
- 接入前准备
- 开发指引

4.5 API字典

- Native下单
- 支付通知
- 微信支付订单号查询订单
- 关闭订单

5.创建项目

5.1 前台项目

5.1.1 安装依赖

```
cra wechatpay-client
cd wechatpay-client
npm install react-router-dom antd axios  react-qrcode-logo
```

包名	介绍
`antd`	Ant Design（简称 antd）是一个由蚂蚁金服开发并维护的基于 React 的 UI 设计语言和框架。它提供了一系列高质量的 React 组件，非常适合开发和服务于企业级后台产品。
`axios`	Axios 是一个基于 Promise 的 HTTP 客户端，用于浏览器和 node.js 环境。它提供了简洁的 API，用于发送请求和处理响应，支持请求拦截、响应拦截、客户端防御 XSRF 等功能。
`react-qrcode-logo`	`react-qrcode-logo` 是一个用于在 React 应用中生成带有自定义 logo 的 QR 码的库。它允许在生成的二维码中添加一个中心 logo，并可以自定义多种属性，如二维码的颜色、尺寸和错误纠正级别。
`react-router-dom`	`react-router-dom` 是一个用于在 React 应用中实现前端路由功能的库，它允许你在应用中创建动态页面导航。

5.2 后台项目

5.2.1 安装依赖

```
mkdir  wechatpay-server
cd wechatpay-server
npm install @fidm/x509  axios  cors dotenv express log4js  mongoose morgan
swagger-jsdoc swagger-ui-express wechatpay-nodejs-sdk
```

包名	介绍
`@fidm/x509`	用于解析 X.509 证书的 Node.js 库。它提供了读取和解析 X.509 证书的能力，常用于安全和加密相关的应用。
`axios`	Axios 是一个基于 Promise 的 HTTP 客户端，用于浏览器和 Node.js 环境。它提供了简洁的 API，用于发送请求和处理响应，支持请求拦截、响应拦截、客户端防御 XSRF 等功能。
`cors`	CORS（Cross-Origin Resource Sharing）的 Node.js 中间件，用于启用 CORS 功能，允许或限制来自不同源的请求。
`dotenv`	一个简单的 Node.js 包，它可以从 `.env` 文件加载环境变量到 `process.env`。这有助于将配置与代码分离，特别是对于敏感信息。
`express`	Express 是一个简洁而灵活的 Node.js Web 应用框架，提供了一系列强大的功能和中间件，以快速地构建 Web 应用和 API。
`log4js`	为 Node.js 应用提供日志管理的库。它支持多种日志输出方式，如文件、控制台等，并允许自定义日志级别、格式和过滤器。
`mongoose`	Mongoose 是一个 MongoDB 对象建模工具，设计用于在异步环境下工作。它为 MongoDB 提供了直观、基于模式的解决方案来建模应用程序数据。
`morgan`	Morgan 是一个用于 Node.js 的 HTTP 请求日志中间件，广泛应用于使用 Express.js 框架的 Web 应用程序中，用于自动记录请求信息。
`swagger-jsdoc`	一个用于从 JSDoc 注释中自动生成 OpenAPI（Swagger）规范的 Node.js 库。这有助于自动化 API 文档的生成过程。
`swagger-ui-express`	用于在 Express 应用程序中集成 Swagger UI 的中间件。这使得 Express 应用可以有一个交互式的 API 文档页面，便于 API 的测试和文档查看。

5.2.2 app.js

wechatpay-server\app.js

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
const app = express();
app.use(morgan('dev'));
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.get('/', async (req, res) => {
    res.send('wechatpay');
});
app.use((error, req, res, next) => {
    logger.error('Error:', error);
    res.status(500).send('Internal Server Error');
});
app.use((req, res) => {
    res.status(404).send('404: Page not found');
});
```

```
module.exports = app;
```

5.2.2 www.js

wechatpay-server\bin\www.js

```
#!/usr/bin/env node
const config = require('../config');
const app = require('../app');
const port = config.PORT || 8000;
app.listen(port,()=>console.log(`Server is running on port ${port}`));
```

5.2.3 .env

wechatpay-server.env

```
PORT=8000
DB_URL=mongodb://127.0.0.1/wechatpay
WECHAT_APP_ID=wx1982d8b56eca9e36
WECHAT_MCH_ID=1319829701
WECHAT_NOTIFY_URL=https://7b28-106-39-150-110.ngrok-free.app
SECRET_KEY=zL1982Q4HzEdHAdMB6DdmqUb4Xyhv4Tk
```

5.2.4 config.js

wechatpay-server\config.js

```
require('dotenv').config();
exports.PORT = process.env.PORT;
exports.DB_URL = process.env.DB_URL;
exports.WECHAT_APP_ID = process.env.WECHAT_APP_ID;
exports.WECHAT_MCH_ID = process.env.WECHAT_MCH_ID;
exports.WECHAT_NOTIFY_URL = process.env.WECHAT_NOTIFY_URL;
exports.SECRET_KEY = process.env.SECRET_KEY;
```

6.查看商品列表

6.1 前台项目

6.1.1 src\index.js

wechatpay-client\src\index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { BrowserRouter } from 'react-router-dom';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <BrowserRouter>
        <App />
    </BrowserRouter>
);
```

6.1.2 App.js

wechatpay-client\src\App.js

```
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import { Layout } from 'antd';
import AppHeader from './components/AppHeader';
import Product from './pages/Product';
import Order from './pages/Order';
const { Content } = Layout;
function App() {
    return (
        <Layout>
            <AppHeader />
            <Content style={{ padding: '50px' }}>
                <Routes>
                    <Route path="/" element={<Product />} />
                    <Route path="/orders" element={<Order />} />
                </Routes>
            </Content>
        </Layout>
    );
}
export default App;
```

6.1.3 AppHeader.js

wechatpay-client\src\components\AppHeader.js

```
import React from 'react';
import { useLocation, NavLink } from 'react-router-dom';
import { Layout, Menu } from 'antd';
const { Header } = Layout;
function AppHeader() {
    const location = useLocation();
    const menuItems = [
        { key: '/', label: <NavLink to="/">商品列表</NavLink> },
        { key: '/orders', label: <NavLink to="/orders">订单管理</NavLink> },
    ];
    return (
        <Header>
            <Menu theme="dark" mode="horizontal" selectedKeys=
{[location.pathname]} items={menuItems} />
        </Header>
    );
}
export default AppHeader;
```

6.1.4 Product.js

wechatpay-client\src\pages\Product.js

```
import React, { useState, useEffect } from 'react';
import { Card, Button } from 'antd';
import api from '../api';
const { Meta } = Card;
function Product() {
    const [products, setProducts] = useState([]);
    const fetchProducts = () => {
        api.get('/api/products').then(response => {
            setProducts(response.data);
        });
    }
    useEffect(fetchProducts, []);
    return (
        <>
            {products.map(product => (
                <Card
                    key={product.id}
                    style={{ width: 240, float: 'left', margin: '16px' }}
                    hoverable
                    cover={<img alt={product.name} src={product.cover} />}
                    actions={[<Button type="primary">购买</Button>]}
                >
                    <Meta title={product.name} description={`价格:
${product.price}`} />
                </Card>
            ))}
        </>
    );
}
export default Product;
```

6.1.5 Order.js

wechatpay-client\src\pages\Order.js

```
import React from 'react';
function Order() {
  return (
    <div>Order</div>
  );
}
export default Order;
```

6.1.6 api\index.js

wechatpay-client\src\api\index.js

```
import axios from 'axios';
const api = axios.create({
    baseURL: 'https://7b28-106-39-150-110.ngrok-free.app',
    headers: { 'ngrok-skip-browser-warning': '69420' }
});
export default api;
```

-

6.2 后台项目

6.2.1 app.js

wechatpay-server\app.js

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
+require('./models');
+const productRoutes = require('./routes/productRoutes');
+const orderRoutes = require('./routes/orderRoutes');
const app = express();
app.use(morgan('dev'));
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.get('/', async (req, res) => {
    res.send('wechatpay');
});
+require('./swagger')(app);
+app.use('/api/products', productRoutes);
+app.use('/api/orders', orderRoutes);
app.use((error, req, res, next) => {
    logger.error('Error:', error);
    res.status(500).send('Internal Server Error');
});
app.use((req, res) => {
    res.status(404).send('404: Page not found');
});
module.exports = app;
```

6.2.2 logger.js

wechatpay-server\logger.js

```
// 引入 log4js 模块
const log4js = require("log4js");
// 获取当前日期和时间
const now = new Date();
// 配置 log4js 的日志记录器
log4js.configure({
    // 定义日志记录的输出方式
    appenders: {
      // 输出到控制台
```

```
      out: { type: 'stdout' },
      // 输出到文件，并根据当前时间命名文件
      fileAppender:  { type: 'file', filename: `logs/log-${now.getHours()+"-
"+now.getMinutes()}.log` }
    },
    // 定义日志类别
    categories: {
      // 默认类别配置为同时输出到控制台和文件，并设置日志级别为 info
      default: { appenders: ['out','fileAppender'], level: 'info' }
    }
});
// 获取一个日志记录器实例
const logger = log4js.getLogger();
// 设置日志记录器的级别为 debug
logger.level = "debug";
// 导出日志记录器，以便在其他模块中使用
module.exports = logger;
```

6.2.3 swagger.js

wechatpay-server\swagger.js

```
// 引入所需的模块
const swaggerJsdoc = require('swagger-jsdoc'); // 用于创建 Swagger 文档对象
const swaggerUi = require('swagger-ui-express'); // 提供了展示 API 文档的 UI 界面
const config = require('./config'); // 引入配置文件
// 获取服务器端口，如果没有在配置文件中指定，则使用默认值 8000
const port = config.PORT || 8000;
// 设置 Swagger 文档的基础信息
const swaggerDefinition = {
    openapi: '3.0.0', // 指定使用 OpenAPI 3.0 规范
    info: {
        title: '微信支付API', // 文档标题
        version: '1.0.0', // 文档版本
        description: '这是微信支付API的API文档', // 文档描述
    },
    tags: [ // 定义 API 标签，用于分类管理不同的 API
        {
            name: 'Product', // 标签名
            description: '产品管理' // 标签描述
        },
        {
            name: 'Payment',
            description: '支付管理'
        },
        {
            name: 'Order',
            description: '订单管理'
        }
    ],
    servers: [ // 定义服务器列表
        {
            url: `http://localhost:${port}`, // 本地服务器的 URL
        },
```

```
    ],
};
// 配置 Swagger 选项
const options = {
    swaggerDefinition, // 前面定义的基础信息
    apis: ['./routes/*.js'], // 指定包含 API 注释的文件路径
};
// 使用配置选项创建 Swagger 文档规范对象
const swaggerSpec = swaggerJsdoc(options);
// 导出一个函数，用于集成 Swagger UI 到 Express 应用
module.exports = (app) => {
    // 在指定路径上设置 Swagger UI
    //swaggerUi.serve: 这是一个中间件函数，提供了必要的静态资源来展示 Swagger UI 页面
    //当访问 /api-docs 路径时，swaggerUi.serve 会处理对 Swagger UI 静态文件（如 HTML, CSS, JavaScript 文件等）的请求
    //swaggerUi.setup(swaggerSpec): 这也是一个中间件函数。它用于配置和初始化 Swagger UI 页面。swaggerUi.setup 接受一个 Swagger 规范对象（在这个例子中是 swaggerSpec），并使用这个对象来生成 API 文档的用户界面
    app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
}
```

6.2.4 productRoutes.js

wechatpay-server\routes\productRoutes.js

```
const express = require('express');
const Product = require('../models/Product');
const router = express.Router();
/**
 * @swagger
 * /api/products:
 *   post:
 *     tags:
 *       - Product
 *     summary: 创建新产品
 *     description: 添加一个新产品到产品列表。
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               name:
 *                 type: string
 *               price:
 *                 type: number
 *     responses:
 *       201:
 *         description: 创建成功，返回新创建的产品
 *       400:
 *         description: 创建失败
 */
router.post('/', async (req, res) => {
    try {
```

```
        const product = new Product(req.body);
        await product.save();
        res.status(201).send(product);
    } catch (error) {
        res.status(400).send(error);
    }
});
/**
 * @swagger
 * /api/products:
 *   get:
 *     tags:
 *       - Product
 *     summary: 获取产品列表
 *     description: 检索产品列表。
 *     responses:
 *       200:
 *         description: 产品列表
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 type: object
 *                 properties:
 *                   name:
 *                     type: string
 *                     description: 产品名称
 *                   price:
 *                     type: number
 *                     description: 产品价格
 */
router.get('/', async (req, res) => {
    try {
        const products = await Product.find({}).sort({ _id: -1 });
        res.send(products);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
 * /api/products/{id}:
 *   get:
 *     tags:
 *       - Product
 *     summary: 获取单个产品
 *     description: 根据 ID 检索单个产品。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 产品ID
 *         schema:
 *           type: string
 *     responses:
```

```
 *       200:
 *         description: 返回请求的产品
 *       404:
 *         description: 未找到产品
 *       500:
 *         description: 服务器错误
 */
router.get('/:id', async (req, res) => {
    try {
        const product = await Product.findById(req.params.id);
        if (!product) {
            return res.status(404).send();
        }
        res.send(product);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
 * /api/products/{id}:
 *   put:
 *     tags:
 *       - Product
 *     summary: 更新产品信息
 *     description: 根据 ID 更新产品信息。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 产品ID
 *         schema:
 *           type: string
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               name:
 *                 type: string
 *               price:
 *                 type: number
 *     responses:
 *       200:
 *         description: 更新成功，返回更新后的产品
 *       400:
 *         description: 更新失败
 *       404:
 *         description: 未找到产品
 */
router.put('/:id', async (req, res) => {
    try {
        const product = await Product.findByIdAndUpdate(req.params.id, req.body,
{ new: true, runValidators: true });
```

```
        if (!product) {
            return res.status(404).send();
        }
        res.send(product);
    } catch (error) {
        res.status(400).send(error);
    }
});
/**
 * @swagger
 * /api/products/{id}:
 *   delete:
 *     tags:
 *       - Product
 *     summary: 删除单个产品
 *     description: 根据 ID 删除单个产品。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 产品ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 删除成功
 *       404:
 *         description: 未找到产品
 *       500:
 *         description: 服务器错误
 */
router.delete('/:id', async (req, res) => {
    try {
        const product = await Product.findByIdAndDelete(req.params.id);
        if (!product) {
            return res.status(404).send();
        }
        res.send(product);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
 * /api/products:
 *   delete:
 *     tags:
 *       - Product
 *     summary: 删除所有产品
 *     description: 删除所有产品。
 *     responses:
 *       200:
 *         description: 删除成功
 *       404:
 *         description: 未找到产品
 *       500:
```

```
 *         description: 服务器错误
 */
router.delete('/', async (req, res) => {
    try {
        const product = await Product.deleteMany({});
        if (!product) {
            return res.status(404).send();
        }
        res.send(product);
    } catch (error) {
        res.status(500).send(error);
    }
});
module.exports = router;
```

6.2.5 orderRoutes.js

wechatpay-server\routes\orderRoutes.js

```
const express = require('express');
const Order = require('../models/Order');
const router = express.Router();
/**
 * @swagger
 * /orders:
 *   post:
 *     tags:
 *       - Order
 *     summary: 创建一个新订单
 *     description: 添加一个新订单到数据库。
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Order'
 *     responses:
 *       201:
 *         description: 创建成功，返回新创建的订单
 *       400:
 *         description: 请求错误
 */
router.post('/', async (req, res) => {
    try {
        const order = new Order(req.body);
        await order.save();
        res.status(201).send(order);
    } catch (error) {
        res.status(400).send(error);
    }
});
/**
 * @swagger
 * /orders:
```

```
 *   get:
 *     tags:
 *       - Order
 *     summary: 获取所有订单
 *     description: 从数据库中检索所有订单。
 *     responses:
 *       200:
 *         description: 成功检索所有订单
 *       500:
 *         description: 服务器错误
 */
router.get('/', async (req, res) => {
    try {
        const orders = await Order.find({}).populate('product').sort({ _id: -1
});
        res.send(orders);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
 * /orders/{id}:
 *   get:
 *     tags:
 *       - Order
 *     summary: 获取特定订单
 *     description: 根据订单ID获取特定订单。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 订单ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 成功检索订单
 *       404:
 *         description: 未找到订单
 *       500:
 *         description: 服务器错误
 */
router.get('/\/:id', async (req, res) => {
    try {
        const order = await Order.findById(req.params.id).populate('product');
        if (!order) {
            return res.status(404).send();
        }
        res.send(order);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
```

```
 * /orders/{id}:
 *   put:
 *     tags:
 *       - Order
 *     summary: 更新特定订单
 *     description: 根据订单ID更新订单信息。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 订单ID
 *         schema:
 *           type: string
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/Order'
 *     responses:
 *       200:
 *         description: 成功更新订单
 *       400:
 *         description: 请求错误
 *       404:
 *         description: 未找到订单
 */
router.put('/:id', async (req, res) => {
    try {
        const order = await Order.findByIdAndUpdate(req.params.id, req.body, {
new: true, runValidators: true });
        if (!order) {
            return res.status(404).send();
        }
        res.send(order);
    } catch (error) {
        res.status(400).send(error);
    }
});
/**
 * @swagger
 * /orders/{id}:
 *   delete:
 *     tags:
 *       - Order
 *     summary: 删除特定订单
 *     description: 根据订单ID删除特定订单。
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         description: 订单ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
```

```
 *         description: 成功删除订单
 *       404:
 *         description: 未找到订单
 *       500:
 *         description: 服务器错误
 */
router.delete('/:id', async (req, res) => {
    try {
        const order = await Order.findByIdAndDelete(req.params.id);
        if (!order) {
            return res.status(404).send();
        }
        res.send(order);
    } catch (error) {
        res.status(500).send(error);
    }
});
/**
 * @swagger
 * /orders:
 *   delete:
 *     tags:
 *       - Order
 *     summary: 删除所有订单
 *     description: 删除数据库中的所有订单。
 *     responses:
 *       200:
 *         description: 成功删除所有订单
 *       500:
 *         description: 服务器错误
 */
router.delete('/', async (req, res) => {
    try {
        const result = await Order.deleteMany({});
        res.send(result);
    } catch (error) {
        res.status(500).send(error);
    }
});
module.exports = router;
```

6.2.6 models\index.js

wechatpay-server\models\index.js

```
const mongoose = require('mongoose');
const {DB_URL} = require('../config');
const logger = require('../logger');
(async function connectToMongoDB() {
    try {
        await mongoose.connect(DB_URL);
        logger.info('MongoDB 连接成功');
    } catch (error) {
        logger.error('MongoDB 连接失败:'+ error.message);
    }
})()
```

6.2.7 Order.js

wechatpay-server\models\Order.js

```
const mongoose = require('mongoose');
const OrderSchema = new mongoose.Schema({
  product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product' },
  totalFee: { type: Number },
  code_url: { type: String },
  orderStatus: { type: String }
},{timestamps: true});
OrderSchema.virtual('id').get(function() {
  return this._id.toHexString();
});
OrderSchema.set('toJSON', {
  virtuals: true
});
OrderSchema.set('toObject', {
  virtuals: true
});
const Order = mongoose.model('Order', OrderSchema);
module.exports = Order;
```

6.2.8 Product.js

wechatpay-server\models\Product.js

```
const mongoose = require('mongoose');
const ProductSchema = new mongoose.Schema({
  name: { type: String, default: '' },
  cover: { type: String, default: '' },
  price: { type: Number, default: 1 },
},{timestamps: true});
ProductSchema.virtual('id').get(function() {
  return this._id.toHexString();
});
ProductSchema.set('toJSON', {
  virtuals: true
});
ProductSchema.set('toObject', {
  virtuals: true
});
```

```
const Product = mongoose.model('Product', ProductSchema);
Product.countDocuments().then(count => {
  if (count === 0) {
    Product.create({
      name: '小米手机',
      cover: 'https://img.yzcdn.cn/vant/ipad.jpeg',
      price: 1
    });
    Product.create({
      name: '华为手机',
      cover: 'https://img.yzcdn.cn/vant/ipad.jpeg',
      price: 2
    });
    Product.create({
      name: '苹果手机',
      cover: 'https://img.yzcdn.cn/vant/ipad.jpeg',
      price: 3
    });
  }
});
module.exports = Product;
```

7.扫码支付

7.1 前台项目

7.1.1 Product.js

wechatpay-client\src\pages\Product.js

```
import React, { useState, useEffect } from 'react';
+import { Card, Button ,Modal } from 'antd';
+import { useNavigate } from 'react-router-dom';
import api from '../api';
+import { scanQRCodePay } from '../helpers';
const { Meta } = Card;
function Product() {
+   const navigate = useNavigate();
    const [products, setProducts] = useState([]);
    const fetchProducts = () => {
        api.get('/api/products').then(response => {
            setProducts(response.data);
        });
    }
    useEffect(fetchProducts, []);
+   async function handleBuy(product) {
+       try {
+           const response = await api.get(`/api/payment/native/${product.id}`);
+           const { code_url, orderNo } = response.data;
+           scanQRCodePay({ orderNo, code_url,callback:()=>navigate('/orders')
});
+       } catch (error) {
+           console.error('下单失败:', error);
+           Modal.error({
+               title: '错误',
```

```
+                content: '无法下单，请稍后重试',
+            });
+        }
+    }
    return (
        <>
            {products.map(product => (
                <Card
                    key={product.id}
                    style={{ width: 240, float: 'left', margin: '16px' }}
                    hoverable
                    cover={<img alt={product.name} src={product.cover} />}
+                   actions={[<Button type="primary" onClick={() =>
handleBuy(product)}>购买</Button>]}
                >
                    <Meta title={product.name} description={`价格:
${product.price}`} />
                </Card>
            ))}
        </>
    );
}
export default Product;
```

7.1.2 helpers.js

wechatpay-client\src\helpers.js

```
import { Modal } from 'antd';
import { QRCode } from 'react-qrcode-logo';
function scanQRCodePay({ code_url,orderNo, callback }) {
  let interval;
  function stopWaiting() {
    clearInterval(interval);
    modal.destroy();
    callback?.();
  }
  const modal = Modal.info({
    title: '支付',
    content: <QRCode value={code_url} />,
    width: 280,
    okText: '取消',
    onOk: () => {
      stopWaiting();
    }
  });
}
export { scanQRCodePay };
```

7.2 后台项目

7.2.1 app.js

wechatpay-server\app.js

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
require('./models');
const productRoutes = require('./routes/productRoutes');
const orderRoutes = require('./routes/orderRoutes');
+const paymentRoutes = require('./routes/paymentRoutes');
const app = express();
app.use(morgan('dev'));
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.get('/', async (req, res) => {
    res.send('wechatpay');
});
require('./swagger')(app);
app.use('/api/products', productRoutes);
app.use('/api/orders', orderRoutes);
+app.use('/api/payment', paymentRoutes);
app.use((error, req, res, next) => {
    logger.error('Error:', error);
    res.status(500).send('Internal Server Error');
});
app.use((req, res) => {
    res.status(404).send('404: Page not found');
});
module.exports = app;
```

7.2.2 constants.js

wechatpay-server\constants.js

```
exports.ORDER_STATUS = {
    UNPAID: 'UNPAID',
    PAID: 'PAID',
    CLOSED: 'CLOSED',
}
```

7.2.3 paymentRoutes.js

wechatpay-server\routes\paymentRoutes.js

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const Product = require('../models/Product');
const Order = require('../models/Order');
const logger = require('../logger');
```

```
//const WechatPay = require('wechatpay-nodejs-sdk');
const WechatPay = require('../wechatpay-nodejs-sdk');
const router = express.Router();
const { ORDER_STATUS } = require('../constants');
const { WECHAT_NOTIFY_URL } = require('../config');
const { WECHAT_APP_ID, WECHAT_MCH_ID, SECRET_KEY } = require('../config');
const wechatPay = new WechatPay({
    appid: WECHAT_APP_ID,
    mchid: WECHAT_MCH_ID,
    publicKey: fs.readFileSync('./apiclient_cert.pem'),
    privateKey: fs.readFileSync('./apiclient_key.pem'),
    secretKey:SECRET_KEY
});
async function createOrder(productId) {
    const product = await Product.findById(productId);
    if (!product) {
        throw new Error('Product not found');
    }
    const newOrder = new Order({
        product: productId,
        totalFee: product.price,
        orderStatus: ORDER_STATUS.UNPAID
    });
    await newOrder.save();
    return { newOrder, product };
}
async function initiateWechatPay(order, product, req) {
    const result = await wechatPay.transactions_native({
        description: `购买${product.name}`,
        out_trade_no: order.id,
        notify_url: `${WECHAT_NOTIFY_URL}/api/payment/callback`,
        amount: { total: product.price },
        scene_info: { payer_client_ip: req.ip },
    });
    logger.info(`wechatPay.transactions_native.result:
${JSON.stringify(result)}`);
    const { code_url } = result;
    await Order.findByIdAndUpdate(order.id, { code_url });
    return code_url;
}
/**
 * @swagger
 * /api/payment/native/{productId}:
 *   get:
 *     tags:
 *       - Payment
 *     summary: 发起微信支付原生交易
 *     description: 为指定产品创建新订单并发起微信支付原生交易。
 *     parameters:
 *       - in: path
 *         name: productId
 *         required: true
 *         description: 产品的唯一ID
 *         schema:
 *           type: string
 *     responses:
```

```
 *       200:
 *         description: 成功发起支付交易
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code_url:
 *                   type: string
 *                   description: 微信支付二维码URL
 *                 orderNo:
 *                   type: string
 *                   description: 订单ID
 *       404:
 *         description: 产品未找到
 *       500:
 *         description: 服务器内部错误
 */
router.get('/native/:productId', async (req, res) => {
    try {
        const { productId } = req.params;
        const { newOrder, product } = await createOrder(productId);
        const code_url = await initiateWechatPay(newOrder, product, req);
        res.send({ code_url, orderNo: newOrder.id });
    } catch (error) {
        logger.error('Native Pay Error:', error);
        res.status(error.message === 'Product not found' ? 404 :
500).send(error.message);
    }
});
module.exports = router;
```

- appid: 微信公众号或小程序的应用ID，这是在微信开放平台注册应用时获得的。
- mchid: 微信支付商户号，这是在微信支付商户平台注册时获得的。
- publicKey: 公钥文件，用于验证微信支付的签名。`fs.readFileSync('./apiclient_cert.pem')` 表示从文件系统中读取名为 apiclient_cert.- pem 的公钥文件。
- privateKey: 私钥文件，用于生成对微信支付请求的签名。`fs.readFileSync('./apiclient_key.pem')` 表示从文件系统中读取名为 - apiclient_key.pem 的私钥文件。
- secretKey: 用于加密的密钥，这是微信支付的一个安全特性，用于确保交易数据的安全性。

7.2.4 wechatpay-nodejs-sdk.js

wechatpay-server\wechatpay-nodejs-sdk.js

```
// 引入需要的模块
const { Certificate } = require('@fidm/x509'); // 用于处理X.509证书
const axios = require('axios'); // 用于发起 HTTP 请求
const crypto = require('crypto'); // 用于加密功能
const DEFAULT_AUTH_TYPE = 'WECHATPAY2-SHA256-RSA2048'; // 默认的认证类型
// 创建一个 axios 实例，预设了微信支付 API 的基础 URL 和默认头部
const weixinPayAPI = axios.create({
```

```
  baseURL: 'https://api.mch.weixin.qq.com',
  headers: { 'Accept': 'application/json', 'Content-Type': 'application/json' }
});
// 获取证书序列号的函数
function getSerialNo(publicKey) {
  return Certificate.fromPEM(publicKey).serialNumber;
}
// 定义 WechatPay 类
class WechatPay {
  // 类的构造函数
  constructor({ appid, mchid, publicKey, privateKey, secretKey, authType }) {
    // 初始化实例变量
    this.appid = appid; // 应用ID
    this.mchid = mchid; // 商户ID
    this.publicKey = publicKey; // 公钥
    this.privateKey = privateKey; // 私钥
    this.secretKey = secretKey; // 密钥
    this.authType = authType || DEFAULT_AUTH_TYPE; // 认证类型
    this.serial_no = getSerialNo(this.publicKey); // 证书序列号
  }
  // 发起请求的异步方法
  async request(method, url, body = {}) {
    // 生成请求所需的随机字符串和时间戳
    const nonce_str = Math.random().toString(36).substring(2, 17);
    const timestamp = Math.floor(Date.now() / 1000).toString();
    // 生成签名
    const signature = this.sign(method, url, nonce_str, timestamp, body);
    // 设置请求头部
    const headers = {
      Authorization: `${this.authType}
mchid="${this.mchid}",nonce_str="${nonce_str}",timestamp="${timestamp}",serial_no
="${this.serial_no}",signature="${signature}"`,
    };
    // 发起请求并返回响应数据
    const responseData = await weixinPayAPI.request({ method, url, data: body,
headers });
    return responseData.data;
  }
  // 签名的方法
  sign(method, url, nonce_str, timestamp, body) {
    // 准备签名所需的数据
    let data = `${method}\n${url}\n${timestamp}\n${nonce_str}\n`;
    data += (method !== 'GET' && body) ? `${JSON.stringify(body)}\n` : '\n';
    // 使用 RSA-SHA256 算法创建并更新签名
    const sign = crypto.createSign('RSA-SHA256');
    sign.update(data);
    // 返回 base64 编码的签名
    return sign.sign(this.privateKey, 'base64');
  }
  // 发起原生支付请求的方法
  async transactions_native(params) {
    const url = '/v3/pay/transactions/native';
    // 准备请求参数
    const requestParams = {
      appid: this.appid,
      mchid: this.mchid,
```

```
      ...params
    };
    // 调用 request 方法发起 POST 请求
    return await this.request('POST', url, requestParams);
  }
}
// 导出 WechatPay 类
module.exports = WechatPay;
```

8.接收支付回调通知

8.1 前台项目

8.2 后台项目

8.2.1 paymentRoutes.js

wechatpay-server\routes\paymentRoutes.js

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const Product = require('../models/Product');
const Order = require('../models/Order');
const logger = require('../logger');
//const WechatPay = require('wechatpay-nodejs-sdk');
const WechatPay = require('../wechatpay-nodejs-sdk');
const router = express.Router();
const { ORDER_STATUS } = require('../constants');
const { WECHAT_NOTIFY_URL } = require('../config');
const { WECHAT_APP_ID, WECHAT_MCH_ID, SECRET_KEY } = require('../config');
const wechatPay = new WechatPay({
    appid: WECHAT_APP_ID,
    mchid: WECHAT_MCH_ID,
    publicKey: fs.readFileSync('./apiclient_cert.pem'),
    privateKey: fs.readFileSync('./apiclient_key.pem'),
    secretKey:SECRET_KEY
  });
async function createOrder(productId) {
    const product = await Product.findById(productId);
    if (!product) {
        throw new Error('Product not found');
    }
    const newOrder = new Order({
        product: productId,
        totalFee: product.price,
        orderStatus: ORDER_STATUS.UNPAID
    });
    await newOrder.save();
    return { newOrder, product };
}
async function initiateWechatPay(order, product, req) {
    const result = await wechatPay.transactions_native({
        description: `购买${product.name}`,
```

```
        out_trade_no: order.id,
        notify_url: `${WECHAT_NOTIFY_URL}/api/payment/callback`,
        amount: { total: product.price },
        scene_info: { payer_client_ip: req.ip },
    });
    logger.info(`wechatPay.transactions_native.result:
${JSON.stringify(result)}`);
    const { code_url } = result;
    await Order.findByIdAndUpdate(order.id, { code_url });
    return code_url;
}
/**
 * @swagger
 * /api/payment/native/{productId}:
 *   get:
 *     tags:
 *       - Payment
 *     summary: 发起微信支付原生交易
 *     description: 为指定产品创建新订单并发起微信支付原生交易。
 *     parameters:
 *       - in: path
 *         name: productId
 *         required: true
 *         description: 产品的唯一ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 成功发起支付交易
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code_url:
 *                   type: string
 *                   description: 微信支付二维码URL
 *                 orderNo:
 *                   type: string
 *                   description: 订单ID
 *       404:
 *         description: 产品未找到
 *       500:
 *         description: 服务器内部错误
 */
router.get('/native/:productId', async (req, res) => {
    try {
        const { productId } = req.params;
        const { newOrder, product } = await createOrder(productId);
        const code_url = await initiateWechatPay(newOrder, product, req);
        res.send({ code_url, orderNo: newOrder.id });
    } catch (error) {
        logger.error('Native Pay Error:', error);
        res.status(error.message === 'Product not found' ? 404 :
500).send(error.message);
    }
```

```
});
+/**
+ * @swagger
+ * /api/payment/callback:
+ *   post:
+ *     tags:
+ *       - Payment
+ *     summary: 微信支付回调端点
+ *     description: 接收来自微信支付的支付通知。
+ *     requestBody:
+ *       required: true
+ *       content:
+ *         application/json:
+ *           schema:
+ *             type: object
+ *             properties:
+ *               event_type:
+ *                 type: string
+ *               resource:
+ *                 type: object
+ *     responses:
+ *       200:
+ *         description: 成功处理回调
+ *         content:
+ *           application/json:
+ *             schema:
+ *               type: object
+ *               properties:
+ *                 code:
+ *                   type: string
+ *                 message:
+ *                   type: string
+ *       500:
+ *         description: 处理回调时出错
+ */
+router.post('/callback', async (req, res) => {
+    try {
+        const { headers, body } = req;
+        const isVerified = await wechatPay.verifySign({
+            body,
+            signature: headers['wechatpay-signature'],
+            serial: headers['wechatpay-serial'],
+            nonce: headers['wechatpay-nonce'],
+            timestamp: headers['wechatpay-timestamp'],
+        });
+        logger.info('isVerified:', isVerified);
+        if (isVerified && body?.event_type === 'TRANSACTION.SUCCESS') {
+            const resultStr = wechatPay.decrypt(body.resource);
+            const result = JSON.parse(resultStr);
+            await Order.findByIdAndUpdate(result.out_trade_no, { orderStatus:
ORDER_STATUS.PAID });
+            res.status(200).send({ code: 'SUCCESS', message: 'Payment
successful' });
+        } else {
+            res.status(200).send({ code: 'FAIL', message: 'Payment failed or
incomplete' });
```

```
+        }
+    } catch (error) {
+        logger.error('Callback Error:', error);
+        res.status(500).send('Error processing callback: ' + error.message);
+    }
+});
module.exports = router;
```

8.2.2 wechatpay-nodejs-sdk.js

wechatpay-server\wechatpay-nodejs-sdk.js

```
const { Certificate } = require('@fidm/x509');
const axios = require('axios');
const crypto = require('crypto');
const DEFAULT_AUTH_TYPE = 'WECHATPAY2-SHA256-RSA2048'
const weixinPayAPI = axios.create({
  baseURL: 'https://api.mch.weixin.qq.com',
  headers: { 'Accept': 'application/json', 'Content-Type': 'application/json' }
});
function getSerialNo(publicKey) {
  return Certificate.fromPEM(publicKey).serialNumber;
}
+ const CACHED_CERTIFICATES = {}; // 创建一个缓存对象来存储获取到的证书
class WechatPay {
  constructor({ appid, mchid, publicKey, privateKey, secretKey, authType }) {
    this.appid = appid;
    this.mchid = mchid;
    this.publicKey = publicKey;
    this.privateKey = privateKey;
    this.secretKey = secretKey;
    this.authType = authType || DEFAULT_AUTH_TYPE;
    this.serial_no = getSerialNo(this.publicKey);
  }
  async request(method, url, body = {}) {
    const nonce_str = Math.random().toString(36).substring(2, 17);
    const timestamp = Math.floor(Date.now() / 1000).toString();
    const signature = this.sign(method, url, nonce_str, timestamp, body);
    const headers = {
      Authorization: `${this.authType}
mchid="${this.mchid}",nonce_str="${nonce_str}",timestamp="${timestamp}",serial_no
="${this.serial_no}",signature="${signature}"`,
    };
    const responseData = await weixinPayAPI.request({ method, url, data: body,
headers });
    return responseData.data;
  }
  sign(method, url, nonce_str, timestamp, body) {
    let data = `${method}\n${url}\n${timestamp}\n${nonce_str}\n`;
    data += (method !== 'GET' && body) ? `${JSON.stringify(body)}\n` : '\n';
    const sign = crypto.createSign('RSA-SHA256');
    sign.update(data);
    return sign.sign(this.privateKey, 'base64');
  }
  async transactions_native(params) {
```

```
    const url = '/v3/pay/transactions/native';
    const requestParams = {
      appid: this.appid,
      mchid: this.mchid,
      ...params
    };
    return await this.request('POST', url, requestParams);
  }
+ async fetchWechatPayPublicKey(serial) {
+   const publicKey = CACHED_CERTIFICATES[serial]; // 尝试从缓存中获取公钥
+   if (publicKey) {
+     return publicKey; // 如果缓存中有公钥，直接返回
+   }
+   const url = '/v3/certificates'; // 微信支付API的URL，用于获取公钥
+   const data = await this.request('GET', url); // 使用封装好的请求方法获取数据
+   data.data.forEach(item => { // 遍历返回的数据
+     const certificate = this.decrypt(item.encrypt_certificate); // 解密证书
+     CACHED_CERTIFICATES[item.serial_no] =
Certificate.fromPEM(certificate).publicKey.toPEM(); // 将解密后的公钥存入缓存
+   });
+   return CACHED_CERTIFICATES[serial]; // 返回指定序列号的公钥
+ }
+ async verifySign(params) {
+   const { timestamp, nonce, body, serial, signature } = params; // 从参数中提取所
需信息
+   let publicKey = await this.fetchWechatPayPublicKey(serial); // 获取公钥
+   const bodyStr = JSON.stringify(body); // 将请求体转换为字符串
+   const data = `${timestamp}\n${nonce}\n${bodyStr}\n`; // 组装要验证的数据
+   const verify = crypto.createVerify('RSA-SHA256'); // 创建一个验证对象
+   verify.update(data); // 更新验证数据
+   return verify.verify(publicKey, signature, 'base64'); // 验证签名是否有效
+ }
+ decrypt(encrypted) {
+   const { ciphertext, associated_data, nonce } = encrypted; // 从加密数据中提取所
需信息
+   const encryptedBuffer = Buffer.from(ciphertext, 'base64'); // 将密文转换为
Buffer
+   const authTag = encryptedBuffer.subarray(encryptedBuffer.length - 16); // 提取
认证标签
+   const encryptedData = encryptedBuffer.subarray(0, encryptedBuffer.length -
16); // 获取加密数据
+   const decipher = crypto.createDecipheriv('aes-256-gcm', this.secretKey,
nonce); // 创建解密器
+   decipher.setAuthTag(authTag); // 设置认证标签（Authentication Tag）
+   decipher.setAAD(Buffer.from(associated_data)); // 设置附加认证数据(Additional
Authenticated Data)
+   const decrypted = Buffer.concat([decipher.update(encryptedData),
decipher.final()]); // 解密数据
+   const decryptedString = decrypted.toString('utf8'); // 将解密结果转换为字符串
+   return decryptedString; // 返回解密结果
+ }
}
module.exports = WechatPay;
```

- `Certificate.fromPEM(certificate).publicKey.toPEM();` 这行代码的作用是：将一个 PEM 格式的证书解析为 Certificate 对象，提取其中的公钥，并将这个公钥再转换为 PEM 格式的字符串

9.轮询订单

9.1 前台项目

9.1.1 helpers.js

wechatpay-client\src\helpers.js

```
import { Modal } from 'antd';
import { QRCode } from 'react-qrcode-logo';
+import { TRADE_STATE,MAX_CHECK_COUNT } from './constants';
+import api from './api';
function scanQRCodePay({ code_url,orderNo, callback }) {
  let interval;
  function stopWaiting() {
    clearInterval(interval);
    modal.destroy();
    callback?.();
  }
  const modal = Modal.info({
    title: '支付',
    content: <QRCode value={code_url} />,
    width: 280,
    okText: '取消',
    onOk: () => {
      stopWaiting();
    }
  });
+ let checkCount = 0;
+ interval = setInterval(() => {
+   if (++checkCount > MAX_CHECK_COUNT) {
+     return stopWaiting();
+   }
+   api.get(`/api/payment/order/${orderNo}`).then(res => {
+     const { trade_state } = res.data;
+     if (trade_state !== TRADE_STATE.NOTPAY) {
+       return stopWaiting();
+     }
+   }).catch(error => {
+     console.error('查询订单状态失败:', error);
+     stopWaiting();
+   });
+ }, 1000);
}
export { scanQRCodePay };
```

9.1.2 constants.js

wechatpay-client\src\constants.js

```
export const TRADE_STATE = {
    SUCCESS: 'SUCCESS',
    NOTPAY: 'NOTPAY'
};
export const MAX_CHECK_COUNT = 30;
```

9.2 后台项目

9.2.1 paymentRoutes.js

wechatpay-server\routes\paymentRoutes.js

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const Product = require('../models/Product');
const Order = require('../models/Order');
const logger = require('../logger');
//const WechatPay = require('wechatpay-nodejs-sdk');
const WechatPay = require('../wechatpay-nodejs-sdk');
const router = express.Router();
const { ORDER_STATUS } = require('../constants');
const { WECHAT_NOTIFY_URL } = require('../config');
const { WECHAT_APP_ID, WECHAT_MCH_ID, SECRET_KEY } = require('../config');
const wechatPay = new WechatPay({
    appid: WECHAT_APP_ID,
    mchid: WECHAT_MCH_ID,
    publicKey: fs.readFileSync('./apiclient_cert.pem'),
    privateKey: fs.readFileSync('./apiclient_key.pem'),
    secretKey:SECRET_KEY
  });
async function createOrder(productId) {
    const product = await Product.findById(productId);
    if (!product) {
        throw new Error('Product not found');
    }
    const newOrder = new Order({
        product: productId,
        totalFee: product.price,
        orderStatus: ORDER_STATUS.UNPAID
    });
    await newOrder.save();
    return { newOrder, product };
}
async function initiateWechatPay(order, product, req) {
    const result = await wechatPay.transactions_native({
        description: `购买${product.name}`,
        out_trade_no: order.id,
        notify_url: `${WECHAT_NOTIFY_URL}/api/payment/callback`,
        amount: { total: product.price },
        scene_info: { payer_client_ip: req.ip },
```

```
    });
    logger.info(`wechatPay.transactions_native.result:
${JSON.stringify(result)}`);
    const { code_url } = result;
    await Order.findByIdAndUpdate(order.id, { code_url });
    return code_url;
}
/**
 * @swagger
 * /api/payment/native/{productId}:
 *   get:
 *     tags:
 *       - Payment
 *     summary: 发起微信支付原生交易
 *     description: 为指定产品创建新订单并发起微信支付原生交易。
 *     parameters:
 *       - in: path
 *         name: productId
 *         required: true
 *         description: 产品的唯一ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 成功发起支付交易
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code_url:
 *                   type: string
 *                   description: 微信支付二维码URL
 *                 orderNo:
 *                   type: string
 *                   description: 订单ID
 *       404:
 *         description: 产品未找到
 *       500:
 *         description: 服务器内部错误
 */
router.get('/native/:productId', async (req, res) => {
    try {
        const { productId } = req.params;
        const { newOrder, product } = await createOrder(productId);
        const code_url = await initiateWechatPay(newOrder, product, req);
        res.send({ code_url, orderNo: newOrder.id });
    } catch (error) {
        logger.error('Native Pay Error:', error);
        res.status(error.message === 'Product not found' ? 404 :
500).send(error.message);
    }
});
/**
 * @swagger
 * /api/payment/callback:
```

```
 *   post:
 *     tags:
 *       - Payment
 *     summary: 微信支付回调端点
 *     description: 接收来自微信支付的支付通知。
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               event_type:
 *                 type: string
 *               resource:
 *                 type: object
 *     responses:
 *       200:
 *         description: 成功处理回调
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code:
 *                   type: string
 *                 message:
 *                   type: string
 *       500:
 *         description: 处理回调时出错
 */
router.post('/callback', async (req, res) => {
    try {
        const { headers, body } = req;
        const isVerified = await wechatPay.verifySign({
            body,
            signature: headers['wechatpay-signature'],
            serial: headers['wechatpay-serial'],
            nonce: headers['wechatpay-nonce'],
            timestamp: headers['wechatpay-timestamp'],
        });
        logger.info('isVerified:', isVerified);
        if (isVerified && body?.event_type === 'TRANSACTION.SUCCESS') {
            const resultStr = wechatPay.decrypt(body.resource);
            const result = JSON.parse(resultStr);
            await Order.findByIdAndUpdate(result.out_trade_no, { orderStatus:
ORDER_STATUS.PAID });
            res.status(200).send({ code: 'SUCCESS', message: 'Payment successful'
});
        } else {
            res.status(200).send({ code: 'FAIL', message: 'Payment failed or
incomplete' });
        }
    } catch (error) {
        logger.error('Callback Error:', error);
        res.status(500).send('Error processing callback: ' + error.message);
```

```
    }
});
+/**
+ * @swagger
+ * /api/payment/order/{orderNo}:
+ *   get:
+ *     tags:
+ *       - Payment
+ *     summary: 查询微信支付交易状态
+ *     description: 使用订单号检索特定微信支付交易的状态。
+ *     parameters:
+ *       - in: path
+ *         name: orderNo
+ *         required: true
+ *         description: 唯一订单号
+ *         schema:
+ *           type: string
+ *     responses:
+ *       200:
+ *         description: 成功检索交易状态
+ *       500:
+ *         description: 查询交易时出错
+ */
+router.get('/order/:orderNo', async (req, res) => {
+    try {
+        const { orderNo } = req.params;
+        const result = await wechatPay.query({ out_trade_no: orderNo });
+        logger.info(`wechatPay.query.result: ${JSON.stringify(result)}`);
+        res.status(200).send(result);
+    } catch (error) {
+        logger.error('Error fetching transaction:', error);
+        res.status(500).send('Error querying transaction');
+    }
+});
module.exports = router;
```

9.2.2 wechatpay-nodejs-sdk.js

wechatpay-server\wechatpay-nodejs-sdk.js

```
const { Certificate } = require('@fidm/x509');
const axios = require('axios');
const crypto = require('crypto');
const DEFAULT_AUTH_TYPE = 'WECHATPAY2-SHA256-RSA2048'
const weixinPayAPI = axios.create({
  baseURL: 'https://api.mch.weixin.qq.com',
  headers: { 'Accept': 'application/json', 'Content-Type': 'application/json' }
});
function getSerialNo(publicKey) {
  return Certificate.fromPEM(publicKey).serialNumber;
}
const CACHED_CERTIFICATES = {};
class WechatPay {
  constructor({ appid, mchid, publicKey, privateKey, secretKey, authType }) {
    this.appid = appid;
```

```
    this.mchid = mchid;
    this.publicKey = publicKey;
    this.privateKey = privateKey;
    this.secretKey = secretKey;
    this.authType = authType || DEFAULT_AUTH_TYPE;
    this.serial_no = getSerialNo(this.publicKey);
  }
  async request(method, url, body = {}) {
    const nonce_str = Math.random().toString(36).substring(2, 17);
    const timestamp = Math.floor(Date.now() / 1000).toString();
    const signature = this.sign(method, url, nonce_str, timestamp, body);
    const headers = {
      Authorization: `${this.authType}
mchid="${this.mchid}",nonce_str="${nonce_str}",timestamp="${timestamp}",serial_no
="${this.serial_no}",signature="${signature}"`,
    };
    const responseData = await weixinPayAPI.request({ method, url, data: body,
headers });
    return responseData.data;
  }
  sign(method, url, nonce_str, timestamp, body) {
    let data = `${method}\n${url}\n${timestamp}\n${nonce_str}\n`;
    data += (method !== 'GET' && body) ? `${JSON.stringify(body)}\n` : '\n';
    const sign = crypto.createSign('RSA-SHA256');
    sign.update(data);
    return sign.sign(this.privateKey, 'base64');
  }
  async transactions_native(params) {
    const url = '/v3/pay/transactions/native';
    const requestParams = {
      appid: this.appid,
      mchid: this.mchid,
      ...params
    };
    return await this.request('POST', url, requestParams);
  }
  async fetchWechatPayPublicKey(serial) {
    const publicKey = CACHED_CERTIFICATES[serial];
    if (publicKey) {
      return publicKey;
    }
    const url = '/v3/certificates';
    const data = await this.request('GET', url);
    data.data.forEach(item => {
      const certificate = this.decrypt(item.encrypt_certificate);
      CACHED_CERTIFICATES[item.serial_no] =
Certificate.fromPEM(certificate).publicKey.toPEM();
    });
    return CACHED_CERTIFICATES[serial];
  }
  async verifySign(params) {
    const { timestamp, nonce, body, serial, signature } = params;
    let publicKey = await this.fetchWechatPayPublicKey(serial);
    const bodyStr = JSON.stringify(body);
    const data = `${timestamp}\n${nonce}\n${bodyStr}\n`;
    const verify = crypto.createVerify('RSA-SHA256');
```

```
    verify.update(data);
    return verify.verify(publicKey, signature, 'base64');
  }
  decrypt(encrypted) {
    const { ciphertext, associated_data, nonce } = encrypted;
    const encryptedBuffer = Buffer.from(ciphertext, 'base64');
    const authTag = encryptedBuffer.subarray(encryptedBuffer.length - 16);
    const encryptedData = encryptedBuffer.subarray(0, encryptedBuffer.length -
16);
    const decipher = crypto.createDecipheriv('aes-256-gcm', this.secretKey,
nonce);
    decipher.setAuthTag(authTag);
    decipher.setAAD(Buffer.from(associated_data));
    const decrypted = Buffer.concat([decipher.update(encryptedData),
decipher.final()]);
    const decryptedString = decrypted.toString('utf8');
    return decryptedString;
  }
+ async query(params) {
+   const { out_trade_no } = params;
+   const url = `/v3/pay/transactions/out-trade-no/${out_trade_no}?
mchid=${this.mchid}`;
+   return await this.request('GET', url);
+ }
}
module.exports = WechatPay;
```

10.订单页

10.1 前台项目

10.1.1 Order.js

wechatpay-client\src\pages\Order.js

```
import React, { useState, useEffect } from 'react';
import { Table, Button, Tag, message, Space } from 'antd';
import { ORDER_STATUS } from '../constants';
import api from '../api';
import { scanQRCodePay } from '../helpers';
function Order() {
  const [orders, setOrders] = useState([]);
  const fetchOrders = () => {
    api.get('/api/orders')
      .then(response => {
        setOrders(response.data);
      })
      .catch(error => {
        console.error('获取订单失败:', error);
        message.error('获取订单失败');
      });
  }
  useEffect(fetchOrders, []);
  const columns = [
    { title: '订单号', dataIndex: 'id', key: 'id' },
```

```
    { title: '产品', dataIndex: 'product', key: 'product', render: product =>
product?.name },
    { title: '总费用', dataIndex: 'totalFee', key: 'totalFee', render: (val) =>
<span>￥{(val / 100).toFixed(2)}</span> },
    {
      title: '状态', dataIndex: 'orderStatus', key: 'orderStatus',
      render: (text) => {
        switch (text) {
          case ORDER_STATUS.PAID: return <Tag color={'green'}>支付成功</Tag>;
          case ORDER_STATUS.UNPAID: return <Tag color={'orange'}>未支付</Tag>;
          case ORDER_STATUS.CLOSED: return <Tag color={'red'}>已关闭</Tag>;
          default: return <Tag>{text}</Tag>;
        }
      }
    },
    {
      title: '操作',
      key: 'action',
      render: (_, record) => {
        if (record.orderStatus === ORDER_STATUS.UNPAID) {
          return (
            <Space>
              <Button type="primary" size='small' onClick={() => scanQRCodePay({
orderNo: record.id, code_url: record.code_url,callback:fetchOrders})}>继续支付
</Button>
            </Space>
          )
        } else {
          return null;
        }
      }
    },
  ];
  return (
    <div style={{ padding: 24 }}>
      <Table columns={columns} dataSource={orders} rowKey="id" />
    </div>
  );
}
export default Order;
```

10.1.2 constants.js

wechatpay-client\src\constants.js

```
export const TRADE_STATE = {
    SUCCESS: 'SUCCESS',
    NOTPAY: 'NOTPAY'
};
export const MAX_CHECK_COUNT = 30;
+export const ORDER_STATUS = {
+    UNPAID: 'UNPAID',
+    PAID: 'PAID',
+    CLOSED: 'CLOSED'
+};
```

10.2 后台项目

11.关闭订单

11.1 前台项目

11.1.1 Order.js

wechatpay-client\src\pages\Order.js

```
import React, { useState, useEffect } from 'react';
import { Table, Button, Tag, message, Space } from 'antd';
import { ORDER_STATUS } from '../constants';
import api from '../api';
import { scanQRCodePay } from '../helpers';
function Order() {
  const [orders, setOrders] = useState([]);
  const fetchOrders = () => {
    api.get('/api/orders')
      .then(response => {
        setOrders(response.data);
      })
      .catch(error => {
        console.error('获取订单失败:', error);
        message.error('获取订单失败');
      });
  }
  useEffect(fetchOrders, []);
+ const closeOrder = (orderNo) => {
+   api.get(`/api/payment/order/${orderNo}/close`)
+     .then(() => {
+       message.success('订单已关闭');
+       fetchOrders();
+     })
+     .catch(error => {
+       console.error('关闭订单失败:', error);
+       message.error('关闭订单失败');
+     });
+ };
  const columns = [
    { title: '订单号', dataIndex: 'id', key: 'id' },
    { title: '产品', dataIndex: 'product', key: 'product', render: product =>
product?.name },
    { title: '总费用', dataIndex: 'totalFee', key: 'totalFee', render: (val) =>
<span>￥{(val / 100).toFixed(2)}</span> },
    {
      title: '状态', dataIndex: 'orderStatus', key: 'orderStatus',
      render: (text) => {
        switch (text) {
          case ORDER_STATUS.PAID: return <Tag color={'green'}>支付成功</Tag>;
          case ORDER_STATUS.UNPAID: return <Tag color={'orange'}>未支付</Tag>;
          case ORDER_STATUS.CLOSED: return <Tag color={'red'}>已关闭</Tag>;
          default: return <Tag>{text}</Tag>;
        }
      }
```

```
    },
    {
      title: '操作',
      key: 'action',
      render: (_, record) => {
        if (record.orderStatus === ORDER_STATUS.UNPAID) {
          return (
            <Space>
              <Button type="primary" size='small' onClick={() => scanQRCodePay({
orderNo: record.id, code_url: record.code_url,callback:fetchOrders})}>继续支付
</Button>
+             <Button danger size='small' onClick={() => closeOrder(record.id)}>
关闭订单</Button>
            </Space>
          )
        } else {
          return null;
        }
      }
    },
  ];
  return (
    <div style={{ padding: 24 }}>
      <Table columns={columns} dataSource={orders} rowKey="id" />
    </div>
  );
}
export default Order;
```

11.2 后台项目

11.2.1 paymentRoutes.js

wechatpay-server\routes\paymentRoutes.js

```
const express = require('express');
const fs = require('fs');
const path = require('path');
const Product = require('../models/Product');
const Order = require('../models/Order');
const logger = require('../logger');
//const WechatPay = require('wechatpay-nodejs-sdk');
const WechatPay = require('../wechatpay-nodejs-sdk');
const router = express.Router();
const { ORDER_STATUS } = require('../constants');
const { WECHAT_NOTIFY_URL } = require('../config');
const { WECHAT_APP_ID, WECHAT_MCH_ID, SECRET_KEY } = require('../config');
const wechatPay = new WechatPay({
    appid: WECHAT_APP_ID,
    mchid: WECHAT_MCH_ID,
    publicKey: fs.readFileSync('./apiclient_cert.pem'),
    privateKey: fs.readFileSync('./apiclient_key.pem'),
    secretKey:SECRET_KEY
  });
async function createOrder(productId) {
```

```
    const product = await Product.findById(productId);
    if (!product) {
        throw new Error('Product not found');
    }
    const newOrder = new Order({
        product: productId,
        totalFee: product.price,
        orderStatus: ORDER_STATUS.UNPAID
    });
    await newOrder.save();
    return { newOrder, product };
}
async function initiateWechatPay(order, product, req) {
    const result = await wechatPay.transactions_native({
        description: `购买${product.name}`,
        out_trade_no: order.id,
        notify_url: `${WECHAT_NOTIFY_URL}/api/payment/callback`,
        amount: { total: product.price },
        scene_info: { payer_client_ip: req.ip },
    });
    logger.info(`wechatPay.transactions_native.result:
${JSON.stringify(result)}`);
    const { code_url } = result;
    await Order.findByIdAndUpdate(order.id, { code_url });
    return code_url;
}
/**
 * @swagger
 * /api/payment/native/{productId}:
 *   get:
 *     tags:
 *       - Payment
 *     summary: 发起微信支付原生交易
 *     description: 为指定产品创建新订单并发起微信支付原生交易。
 *     parameters:
 *       - in: path
 *         name: productId
 *         required: true
 *         description: 产品的唯一ID
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 成功发起支付交易
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code_url:
 *                   type: string
 *                   description: 微信支付二维码URL
 *                 orderNo:
 *                   type: string
 *                   description: 订单ID
 *       404:
```

```
 *         description: 产品未找到
 *       500:
 *         description: 服务器内部错误
 */
router.get('/native/:productId', async (req, res) => {
    try {
        const { productId } = req.params;
        const { newOrder, product } = await createOrder(productId);
        const code_url = await initiateWechatPay(newOrder, product, req);
        res.send({ code_url, orderNo: newOrder.id });
    } catch (error) {
        logger.error('Native Pay Error:', error);
        res.status(error.message === 'Product not found' ? 404 :
500).send(error.message);
    }
});
/**
 * @swagger
 * /api/payment/callback:
 *   post:
 *     tags:
 *       - Payment
 *     summary: 微信支付回调端点
 *     description: 接收来自微信支付的支付通知。
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               event_type:
 *                 type: string
 *               resource:
 *                 type: object
 *     responses:
 *       200:
 *         description: 成功处理回调
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 code:
 *                   type: string
 *                 message:
 *                   type: string
 *       500:
 *         description: 处理回调时出错
 */
router.post('/callback', async (req, res) => {
    try {
        const { headers, body } = req;
        const isVerified = await wechatPay.verifySign({
            body,
            signature: headers['wechatpay-signature'],
```

```
            serial: headers['wechatpay-serial'],
            nonce: headers['wechatpay-nonce'],
            timestamp: headers['wechatpay-timestamp'],
        });
        logger.info('isVerified:', isVerified);
        if (isVerified && body?.event_type === 'TRANSACTION.SUCCESS') {
            const resultStr = wechatPay.decrypt(body.resource);
            const result = JSON.parse(resultStr);
            await Order.findByIdAndUpdate(result.out_trade_no, { orderStatus:
ORDER_STATUS.PAID });
            res.status(200).send({ code: 'SUCCESS', message: 'Payment successful'
});
        } else {
            res.status(200).send({ code: 'FAIL', message: 'Payment failed or
incomplete' });
        }
    } catch (error) {
        logger.error('Callback Error:', error);
        res.status(500).send('Error processing callback: ' + error.message);
    }
});
/**
 * @swagger
 * /api/payment/order/{orderNo}:
 *   get:
 *     tags:
 *       - Payment
 *     summary: 查询微信支付交易状态
 *     description: 使用订单号检索特定微信支付交易的状态。
 *     parameters:
 *       - in: path
 *         name: orderNo
 *         required: true
 *         description: 唯一订单号
 *         schema:
 *           type: string
 *     responses:
 *       200:
 *         description: 成功检索交易状态
 *       500:
 *         description: 查询交易时出错
 */
router.get('/order/:orderNo', async (req, res) => {
    try {
        const { orderNo } = req.params;
        const result = await wechatPay.query({ out_trade_no: orderNo });
        logger.info(`wechatPay.query.result: ${JSON.stringify(result)}`);
        res.status(200).send(result);
    } catch (error) {
        logger.error('Error fetching transaction:', error);
        res.status(500).send('Error querying transaction');
    }
});
+/**
+ * @swagger
+ * /api/payment/order/{orderNo}/close:
```

```
+ *   get:
+ *     tags:
+ *       - Payment
+ *     summary: 关闭微信支付交易
+ *     description: 使用订单号关闭正在进行的微信支付交易。
+ *     parameters:
+ *       - in: path
+ *         name: orderNo
+ *         required: true
+ *         description: 唯一订单号
+ *         schema:
+ *           type: string
+ *     responses:
+ *       200:
+ *         description: 订单成功关闭
+ *       500:
+ *         description: 关闭订单时出错
+ */
+router.get('/order/:orderNo/close', async (req, res) => {
+    try {
+        const { orderNo } = req.params;
+        await wechatPay.close({ out_trade_no: orderNo });
+        await Order.findByIdAndUpdate(orderNo, { orderStatus:
ORDER_STATUS.CLOSED });
+        res.send({ message: 'Order successfully closed' });
+    } catch (error) {
+        logger.error('Error closing order:', error);
+        res.status(500).send('Error closing order');
+    }
+});
module.exports = router;
```

11.2.2 wechatpay-nodejs-sdk.js

wechatpay-server\wechatpay-nodejs-sdk.js

```
const { Certificate } = require('@fidm/x509');
const axios = require('axios');
const crypto = require('crypto');
const DEFAULT_AUTH_TYPE = 'WECHATPAY2-SHA256-RSA2048'
const weixinPayAPI = axios.create({
  baseURL: 'https://api.mch.weixin.qq.com',
  headers: { 'Accept': 'application/json', 'Content-Type': 'application/json' }
});
function getSerialNo(publicKey) {
  return Certificate.fromPEM(publicKey).serialNumber;
}
const CACHED_CERTIFICATES = {};
class WechatPay {
  constructor({ appid, mchid, publicKey, privateKey, secretKey, authType }) {
    this.appid = appid;
    this.mchid = mchid;
    this.publicKey = publicKey;
    this.privateKey = privateKey;
    this.secretKey = secretKey;
```

```
    this.authType = authType || DEFAULT_AUTH_TYPE;
    this.serial_no = getSerialNo(this.publicKey);
  }
  async request(method, url, body = {}) {
    const nonce_str = Math.random().toString(36).substring(2, 17);
    const timestamp = Math.floor(Date.now() / 1000).toString();
    const signature = this.sign(method, url, nonce_str, timestamp, body);
    const headers = {
      Authorization: `${this.authType}
mchid="${this.mchid}",nonce_str="${nonce_str}",timestamp="${timestamp}",serial_no
="${this.serial_no}",signature="${signature}"`,
    };
    const responseData = await weixinPayAPI.request({ method, url, data: body,
headers });
    return responseData.data;
  }
  sign(method, url, nonce_str, timestamp, body) {
    let data = `${method}\n${url}\n${timestamp}\n${nonce_str}\n`;
    data += (method !== 'GET' && body) ? `${JSON.stringify(body)}\n` : '\n';
    const sign = crypto.createSign('RSA-SHA256');
    sign.update(data);
    return sign.sign(this.privateKey, 'base64');
  }
  async transactions_native(params) {
    const url = '/v3/pay/transactions/native';
    const requestParams = {
      appid: this.appid,
      mchid: this.mchid,
      ...params
    };
    return await this.request('POST', url, requestParams);
  }
  async fetchWechatPayPublicKey(serial) {
    const publicKey = CACHED_CERTIFICATES[serial];
    if (publicKey) {
      return publicKey;
    }
    const url = '/v3/certificates';
    const data = await this.request('GET', url);
    data.data.forEach(item => {
      const certificate = this.decrypt(item.encrypt_certificate);
      CACHED_CERTIFICATES[item.serial_no] =
Certificate.fromPEM(certificate).publicKey.toPEM();
    });
    return CACHED_CERTIFICATES[serial];
  }
  async verifySign(params) {
    const { timestamp, nonce, body, serial, signature } = params;
    let publicKey = await this.fetchWechatPayPublicKey(serial);
    const bodyStr = JSON.stringify(body);
    const data = `${timestamp}\n${nonce}\n${bodyStr}\n`;
    const verify = crypto.createVerify('RSA-SHA256');
    verify.update(data);
    return verify.verify(publicKey, signature, 'base64');
  }
  decrypt(encrypted) {
```

```
    const { ciphertext, associated_data, nonce } = encrypted;
    const encryptedBuffer = Buffer.from(ciphertext, 'base64');
    const authTag = encryptedBuffer.subarray(encryptedBuffer.length - 16);
    const encryptedData = encryptedBuffer.subarray(0, encryptedBuffer.length -
16);
    const decipher = crypto.createDecipheriv('aes-256-gcm', this.secretKey,
nonce);
    decipher.setAuthTag(authTag);
    decipher.setAAD(Buffer.from(associated_data));
    const decrypted = Buffer.concat([decipher.update(encryptedData),
decipher.final()]);
    const decryptedString = decrypted.toString('utf8');
    return decryptedString;
  }
  async query(params) {
    const { out_trade_no } = params;
    const url = `/v3/pay/transactions/out-trade-no/${out_trade_no}?
mchid=${this.mchid}`;
    return await this.request('GET', url);
  }
+ async close(params) {
+   const { out_trade_no } = params;
+   const url = `/v3/pay/transactions/out-trade-no/${out_trade_no}/close`;
+   await this.request('POST', url, { mchid: this.mchid });
+ }
}
module.exports = WechatPay;
```

参考

1.virtuals

1. **OrderSchema.set('toJSON', {...})**:
   - 这一行设置了当一个 Mongoose 文档被转换为 JSON 格式时的行为。
   - `'toJSON'`: 这是一个选项，当文档使用 `.toJSON()` 方法转换为 JSON 时会被考虑。
   - `{ virtuals: true }`: 这个配置对象指定在将文档转换为 JSON 时包含虚拟属性（virtuals）。虚拟属性是 Mongoose 中的一种特性，它们不直接存储在数据库中，而是在运行时动态计算得到的属性。通过设置 `virtuals: true`，这些属性会被包含在 JSON 表示中。
2. **OrderSchema.set('toObject', {...})**:
   - 这一行设置了当一个 Mongoose 文档被转换为普通的 JavaScript 对象时的行为。
   - `'toObject'`: 这是一个选项，当文档使用 `.toObject()` 方法转换为一个普通的 JavaScript 对象时会被考虑。
   - `{ virtuals: true }`: 类似于 `toJSON` 的设置，这确保在转换为 JavaScript 对象时，虚拟属性也会被包含。

2.X.509 证书

X.509 是一种广泛使用的标准，用于定义公钥证书的格式。这种证书主要用于建立安全的网络通信，特别是在互联网上。下面是关于 X.509 证书的一些关键点：

基本概念和用途

1. **公钥基础设施（PKI）**：
   - X.509 证书是公钥基础设施（PKI）的一部分。PKI 是一套标准和服务，用于支持大规模使用数字证书和公钥加密。
   - 在 PKI 中，证书用于验证公钥的所有权，确保公钥真的属于声称拥有它的实体。
2. **用途**:
   - X.509 证书广泛用于安全通信，如 HTTPS。
   - 它们用于加密数据、建立安全连接和验证通信双方的身份。

证书结构

一个 X.509 证书包含以下重要部分：

1. **版本号**:
   - 指定证书遵循的 X.509 版本。
2. **序列号**:
   - 证书的唯一标识符。
3. **签名算法**:
   - 用于签名证书的算法。
4. **颁发者名称**:
   - 颁发证书的认证机构（CA）的名称。
5. **有效期**:
   - 证书的有效开始和结束日期。
6. **主体名称**:
   - 证书所有者的身份信息。
7. **公钥**:
   - 证书主体的公钥。
8. **扩展**:
   - 提供关于证书用途和策略的附加信息。

证书颁发和验证

- **颁发**:
  - X.509 证书由认证机构（CA）颁发。CA 是一个受信任的实体，负责验证申请证书者的身份并颁发证书。
- **验证**:
  - 当一个实体（如一个网站）提供其 X.509 证书时，客户端（如浏览器）会验证该证书的有效性。
  - 这包括检查证书是否由受信任的 CA 签名、证书是否在有效期内、以及证书的其他安全属性。

证书链

- 证书通常不是直接由根 CA 颁发的，而是通过一系列中间 CA 创建的。这种结构称为“证书链”。
- 客户端在验证证书时，会沿着证书链一直检查到根 CA，确保证书的完整可信。

总结

X.509 证书是建立和维护数字信任的基础。它们在确保网络通信安全、验证身份和加密数据传输中起着关键作用。理解和正确处理 X.509 证书对于保障网络安全至关重要。

3.PEM

PEM（Privacy-Enhanced Mail）是一种用于存储和发送加密数据的文本格式，主要用于保存加密密钥和证书。尽管其名称源自一个早期的电子邮件加密协议，但今天 PEM 格式在许多不同的加密应用中被广泛使用，特别是在 SSL/TLS 和其它基于 X.509 证书的系统中。以下是关于 PEM 格式的一些关键点：

格式特点

1. **ASCII 编码**:
    - PEM 格式是一种基于文本的编码方式，使用 ASCII 字符集，这使得它可以在不同的系统和程序之间轻松地共享和传输。
2. **Base64 编码**:
    - PEM 格式通常包含 Base64 编码的二进制数据。Base64 是一种编码方法，用于将二进制数据转换为 ASCII 文本。
3. **标头和标尾**:
    - PEM 文件包含明确的开始和结束标记，如 `-----BEGIN CERTIFICATE-----` 和 `-----END CERTIFICATE-----`。这些标记指明了数据的类型（例如证书、私钥等）。

常见用途

- **证书和密钥**:
    - PEM 格式广泛用于存储和传输各种类型的加密对象，包括公钥、私钥、证书、CSR（证书签名请求）等。
- **SSL/TLS**:
    - 在 SSL/TLS 协议中，PEM 格式被用于交换和存储服务器和客户端的证书及私钥。

兼容性

- 由于其简单和文本基础的特性，PEM 格式与多种编程语言和加密库兼容，包括 OpenSSL、Java、Python 等。

安全性

- 尽管 PEM 格式本身不提供加密，但它可以用于存储加密的密钥数据。这种情况下，文件通常会包含额外的密码保护。

示例

一个典型的 PEM 格式的证书看起来像这样：

```
-----BEGIN CERTIFICATE-----
MIID1zCCAr+gAwIBAgIJAMnN0nLbLBrIMA0GCSqGSIb3DQEBCwUAMFoxCzAJBgNV
...（更多 Base64 编码的数据）...
-----END CERTIFICATE-----
```

总结

PEM 是一种方便、灵活的格式，用于处理和传输加密材料。它的文本基础和简明的结构使得它成为加密应用中的一个重要和广泛使用的标准。

4.MongoDB

安装MongodbDB
在Windows上安装MongoDB的步骤如下：

1. **下载MongoDB安装程序**：访问MongoDB下载中心，选择需要的版本、平台（Windows）和安装包（msi），然后点击“下载”。
2. **运行安装程序**：在“下载”文件夹中找到下载的`.msi`文件，双击开始安装。
3. **按照安装向导操作**：安装向导会指导你完成安装过程。你可以选择“完整”安装（推荐大多数用户）或“自定义”安装类型。“完整”安装会将MongoDB及其工具安装到默认位置。
4. **服务配置和MongoDB Compass**：在安装过程中，你可以选择将MongoDB设置为Windows服务。你还可以选择安装MongoDB Compass。
5. **安装MongoDB Shell（`mongosh`）**：请注意，MongoDB Shell（`mongosh`）并不随MongoDB服务器一起安装，需要单独安装。

安装MongoDB Shell
要安装MongoDB Shell（mongosh），请按照以下步骤操作：

1. 访问MongoDB的下载中心。
2. 选择适合你操作系统的MongoDB Shell版本。
3. 下载相应的安装文件。
4. 根据你的操作系统，运行下载的安装程序并按照指示完成安装。

这些步骤会为你的系统安装最新版本的MongoDB Shell，使你能够通过命令行界面与MongoDB数据库进行交互。安装后，你可以通过命令行启动mongosh并开始使用。

MongoDB Compass
MongoDB Compass是一个图形界面工具，用于管理MongoDB数据库。它提供了数据的可视化表示，使用户能够直观地浏览和操作数据库中的数据。Compass支持复杂的查询，数据建模，索引优化，并提供了实时性能分析。它简化了数据库管理任务，允许用户直接在界面上进行数据的增删改查，查看数据结构，运行查询，并查看查询性能分析结果。Compass适合希望通过图形界面而非命令行来管理MongoDB的用户。

5.AES-256-GCM

`AES-256-GCM`是一种加密算法，用于确保数据传输的安全性。它由几个组成部分构成：AES、256位密钥长度、GCM模式。下面是对这些组成部分的详细解释：

AES (Advanced Encryption Standard)

- **AES** 是一种广泛使用的对称加密标准，意味着加密和解密使用相同的密钥。
- 它被认为是一种安全的加密形式，用于保护电子数据。

256位密钥长度

- **256位** 是指用于加密的密钥的长度。AES 支持多种长度的密钥，通常包括 128位、192位和 256 位。
- 256位密钥提供了极强的安全性，是目前可用的最强密钥长度之一。

GCM (Galois/Counter Mode)

- **GCM** 是一种操作模式，它用于 AES 加密。
- GCM 提供了两个重要的功能：
  - **加密**：保证数据的保密性。
  - **认证**：验证数据的完整性和真实性。
- 这种模式特别适合于保护网络数据传输，因为它同时提供了加密和消息认证。

AES-256-GCM 的特点

1. **高安全性**：由于其 256 位的密钥长度，AES-256-GCM 非常难以破解。
2. **性能优良**：GCM 模式支持并行处理，这意味着它可以快速加密和解密数据，特别是在现代处理器上。
3. **数据完整性**：除了加密，GCM 还提供了完整性检查，确保数据在传输过程中没有被篡改。
4. **抗重放攻击**：GCM 使用一个称为“nonce”的计数器，它可以帮助防止重放攻击，即防止同一加密文本被多次发送来欺骗系统。

使用场景

- `AES-256-GCM` 通常用于需要高安全性和数据完整性验证的场景，例如在互联网通信、文件加密、安全协议（如 TLS）中。

结论

总的来说，`AES-256-GCM` 是一种非常强大的加密技术，适用于需要高度安全保障和数据完整性的应用场景。由于其加密和认证的双重功能，它在现代网络安全中扮演着重要角色。

6.AuthTag和AAD

AES-256-GCM 模式是一种对称密钥加密算法，它提供了认证加密的功能。在这种模式下，除了加密数据本身，还生成了一个称为“认证标签”（Authentication Tag）的额外数据，用于保证数据的完整性和真实性。同时，“附加认证数据”（Additional Authenticated Data，AAD）是与密文一起传输但不加密的数据，也用于验证数据的完整性和真实性。

1. **设置认证标签 (AuthTag)**:
   - `decipher.setAuthTag(authTag);`
   - 这行代码用于设置解密过程中的认证标签（AuthTag）。在 AES-GCM 加密模式中，AuthTag 用于验证数据的完整性和真实性。
   - 当解密数据时，AuthTag 需要与加密时生成的 AuthTag 匹配，以确保数据在传输过程中未被篡改。如果不匹配，解密过程将抛出错误，表明数据可能被篡改。
2. **设置附加认证数据 (AAD)**:
   - `decipher.setAAD(Buffer.from(associated_data));`
   - 这行代码设置了附加认证数据（AAD）。AAD 是在加密过程中使用的数据，但与主要要加密的数据分开，AAD 不被加密。
   - 在 GCM 模式中，AAD 也用于生成 AuthTag。因此，在解密时，必须提供与加密时相同的 AAD，以便正确验证 AuthTag。
   - AAD 通常用于传输一些公开的元数据，这些元数据虽然不需要保密，但它们的完整性和真实性需要得到保证。

总的来说，`setAuthTag` 和 `setAAD` 在解密过程中用于确保数据的安全性。通过验证 AuthTag 和 AAD，可以确保密文在传输过程中未被篡改，从而保证数据的完整性和真实性。这对于实现安全的数据通信至关重要。

7.RSA-SHA256

`RSA-SHA256` 是一个在数字签名和加密领域常用的术语，它结合了两种不同的加密技术：RSA 加密算法和 SHA-256 哈希算法。下面分别解释这两种技术及其组合使用时的作用：

RSA-SHA256

1. **RSA 概述**：
   - RSA 是一种非对称加密算法，广泛用于安全数据传输。
   - 在 RSA 中，加密和解密使用一对不同的密钥：一个公钥和一个私钥。
2. **使用场景**：
   - RSA 主要用于加密数据和创建数字签名。
   - 公钥用于加密数据或验证签名，而私钥用于解密或创建签名。

SHA-256 哈希算法

1. **SHA-256 概述**：
   - SHA-256 是一种安全哈希算法，属于 SHA-2 家族。
   - 它输出一个固定大小（256位）的哈希值，通常表示为 64 个十六进制字符。
2. **使用场景**：
   - SHA-256 通常用于生成数据的哈希表示，以确保数据的完整性和一致性。
   - 在数字签名中，它用于创建消息的摘要，然后被加密以生成签名。

RSA-SHA256 的结合使用

1. **数字签名**：
   - 在数字签名的上下文中，`RSA-SHA256` 指的是首先使用 SHA-256 生成消息的摘要，然后用 RSA 私钥加密该摘要以创建签名的过程。
   - 这种签名保证了信息的完整性和发送者的真实性。
2. **验证过程**：
   - 收件人使用相同的 SHA-256 算法对原始消息生成哈希值，然后使用发送者的公钥解密签名以获取摘要。如果两个摘要匹配，证明消息未被篡改且确实来自声称的发送者。

结论

`RSA-SHA256` 结合了 RSA 的加密强度和 SHA-256 的哈希可靠性，是一种在保证数据传输安全和验证数据完整性方面非常有效的技术。这种方法在许多现代安全应用中被广泛使用，如安全电子邮件传输、安全文档签名和SSL/TLS等安全协议。

8.createVerify

1. **创建验证对象**：
   - `const verify = crypto.createVerify('RSA-SHA256');`

- 这行代码使用 `crypto` 模块的 `createVerify` 方法创建一个验证对象。`'RSA-SHA256'` 指定了验证过程中使用的算法，即 RSA 签名算法和 SHA-256 哈希算法。
- 这个验证对象将用于验证一个数字签名是否是用特定的私钥生成的。

2. **更新验证数据**:
   - `verify.update(data);`
   - 这行代码用 `update` 方法向验证对象添加数据。在验证数字签名时，你需要提供原始数据（这里是 `data`），这些数据在签名过程中也是被签名的数据。
   - `data` 应该是一个字符串或者 Buffer，并且这应该是在生成签名时使用的完全相同的数据。
3. **验证签名**:
   - `return verify.verify(publicKey, signature, 'base64');`
   - 这行代码用 `verify` 方法来验证签名的有效性。
   - `publicKey` 是用于验证签名的公钥。如果签名是由对应的私钥正确生成的，那么这个方法将返回 `true`，否则返回 `false`。
   - `signature` 是需要验证的签名，这里假设它是一个 Base64 编码的字符串。
   - `'base64'` 参数指定了签名的格式。在这个例子中，签名是以 Base64 编码的形式提供的。

9. antd

Ant Design（通常简称为 `antd`）是一个基于 React 的 UI 设计框架，专为构建企业级中后台前端应用而设计。它提供了一整套设计语言和高质量的 React 组件。

1.1 主要特点

1. **丰富的组件库**：Ant Design 提供了丰富的 React 组件，涵盖了从基础按钮、输入框到高级表格、表单、导航菜单等几乎所有开发中后台应用所需的组件。
2. **企业级实践**：设计和开发标准符合企业级产品的需求，非常适合用于复杂的商业系统。
3. **高度可定制**：提供主题定制功能，可以很容易地调整组件的样式以符合品牌和设计要求。
4. **国际化支持**：支持多语言，满足不同地区用户的需求。
5. **良好的文档和社区支持**：具有详细的文档和广泛的社区支持，使得开发者容易上手和使用。

应用场景

由于其企业级的特性，Ant Design 主要用于构建功能复杂的中后台系统，如企业管理系统、客户关系管理系统（CRM）、内容管理系统（CMS）等。

1.2 使用方法

在 React 项目中使用 Ant Design 需要先安装相应的 npm 包：

```
npm install antd
```

然后，可以在 React 组件中导入并使用 Ant Design 的组件：

```
import React from 'react';
import { Button } from 'antd';

const App = () => (
  <div>
    <Button type="primary">Primary Button</Button>
  </div>
);

export default App;
```

此外，Ant Design 还提供了一套完整的设计资源和开发工具，如 Ant Design Pro，这是一个构建企业级中后台应用的上手模板，提供了丰富的布局、页面模板和典型页面。

总的来说，Ant Design 是一个功能全面、设计美观且易于使用的 React UI库，非常适合用于构建高质量的企业级应用。

10. axios

`axios` 是一个基于 Promise 的 HTTP 客户端，用于浏览器和 node.js 环境中。它主要用于发送 HTTP 请求和处理 HTTP 响应。

2.1 主要特点

1. **从浏览器创建 XMLHttpRequests**：在浏览器中，`axios` 使用 XMLHttpRequest 对象发送 HTTP 请求。
2. **从 node.js 发送 http 请求**：在 node.js 环境中，它使用 Node.js 的原生 http 模块。
3. **支持 Promise API**：所有请求都返回一个 Promise 对象，可以轻松使用 `then` 和 `catch` 方法处理异步响应。
4. **拦截请求和响应**：在请求或响应被 then 或 catch 处理之前，可以拦截它们，这在处理全局的请求或响应逻辑时非常有用。
5. **转换请求和响应数据**：允许在请求和响应数据发送或接收前对其进行转换。
6. **自动转换 JSON 数据**：发送请求时，如果请求体是 JavaScript 对象，`axios` 会自动将其转换为 JSON 字符串。响应时，如果检测到返回的内容是 JSON 格式，`axios` 也会自动将其转换为 JavaScript 对象。
7. **客户端支持防御 XSRF**：在浏览器端，`axios` 有内置的 XSRF 防护。

2.2 使用示例

以下是使用 `axios` 发送一个简单的 GET 请求的例子：

```
const axios = require('axios');

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

2.3 安装

在使用 `axios` 前，需要先通过 npm 或 yarn 安装它：

```
npm install axios
```

或者

```
yarn add axios
```

2.4 配置全局默认值

`axios` 允许全局配置请求的默认值，如基础 URL、请求头等：

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

2.5 总结

`axios` 提供了一个简单且灵活的方式来发送 HTTP 请求，并且它的 Promise API 使得异步请求的处理变得更加方便。这使得 `axios` 成为开发现代 Web 应用程序和 Node.js 应用程序时一个非常受欢迎的选择。

11. react-qrcode-logo

`react-qrcode-logo` 是一个用于在 React 应用中生成带有自定义 logo 的 QR 码的库。这个库允许你在生成的二维码中添加一个中心logo，并且可以自定义许多其他选项，比如二维码的颜色、尺寸和错误纠正级别。下面是如何在 React 项目中使用 `react-qrcode-logo` 的基本步骤：

1. **安装**: 首先，你需要在你的 React 项目中安装 `react-qrcode-logo`。你可以通过 npm 来安装它：

   ```
   npm install react-qrcode-logo
   ```

   或者使用 yarn：

   ```
   yarn add react-qrcode-logo
   ```

2. **导入并使用组件**: 安装完成后，你可以在你的 React 组件中导入并使用 `QRCode` 组件。例如：

   ```
   import React from 'react';
   import QRCode from 'react-qrcode-logo';

   const MyComponent = () => {
     return (
       <QRCode
         value="https://www.example.com"
         logoImage="path/to/logo.png"
         logoWidth={50}
         logoHeight={50}
       />
     );
   };

   export default MyComponent;
   ```

在这个例子中，`QRCode` 组件创建了一个指向 "https://www.example.com" 的 QR 码，并在中间嵌入了一张 logo 图片。

3. **自定义选项**: `react-qrcode-logo` 允许你自定义多种属性，例如：
   - `value`: 要编码的字符串或 URL。
   - `logoImage`: 中心 logo 的图片路径。
   - `logoWidth` 和 `logoHeight`: Logo 的宽度和高度。
   - `size`: 二维码的尺寸（宽度和高度）。
   - `ecLevel`: 错误纠正级别（'L', 'M', 'Q', 'H'）。
   - `bgColor` 和 `fgColor`: 二维码的背景颜色和前景颜色。

   你可以根据需要调整这些属性来自定义你的二维码。
4. **渲染组件**: 最后，将 `QRCode` 组件放在你的 React 应用的任何位置来渲染二维码。

`react-qrcode-logo` 是一个灵活且易于使用的库，非常适合需要在二维码中嵌入 logo 或者需要特定样式二维码的 React 项目。

12. @fidm/x509

`@fidm/x509` 是一个 Node.js 库，用于解析 X.509 证书。X.509 是一种非常普遍的数字证书标准，广泛用于互联网和电子商务领域，尤其是在 TLS/SSL 协议中用于安全通信。

4.1 主要功能

1. **解析 X.509 证书**：该库能够解析出证书中的各种信息，包括证书的颁发机构、有效期、主题、公钥等。
2. **支持多种格式**：可以解析 DER（二进制格式）和 PEM（Base64 编码格式）。
3. **获取证书指纹**：能够提取证书的指纹，这是证书的一个唯一标识符。
4. **证书验证**：可以用来验证证书的有效性，比如检查证书是否在有效期内，是否被某个可信的证书颁发机构颁发等。

4.2 使用场景

- **TLS/SSL 通信**：在处理安全通信时，解析和验证 TLS/SSL 证书。
- **数字签名验证**：在处理数字签名时，验证签名所依赖的证书。
- **安全分析**：在进行网络安全分析和测试时，检查和分析证书的详细信息。

4.3 示例代码

下面是一个简单的示例，展示了如何使用 `@fidm/x509` 来解析一个 PEM 格式的 X.509 证书：

```
const { Certificate } = require('@fidm/x509');

const pem = '-----BEGIN CERTIFICATE-----\n...\n-----END CERTIFICATE-----'; // 这里应该是您的证书内容
const cert = Certificate.fromPEM(Buffer.from(pem));

console.log('Subject:', cert.subject);
console.log('Issuer:', cert.issuer);
console.log('Valid from:', cert.validFrom);
console.log('Valid to:', cert.validTo);
```

4.4 安装

要开始使用 `@fidm/x509`，您需要先在您的 Node.js 项目中安装它：

```
npm install @fidm/x509
```

4.5 总结

`@fidm/x509` 是一个非常有用的库，用于在 Node.js 应用程序中处理和解析 X.509 证书。它的功能强大，可以满足绝大多数关于数字证书解析和验证的需求。

13. cors

`cors` 是一个非常流行的 Node.js 包，用于在 Express.js（或其他基于 Node.js 的 web 应用框架）中启用 CORS（跨源资源共享）。CORS 是一种安全功能，它允许或限制 web 应用访问不同域（origin）上的资源。默认情况下，浏览器出于安全考虑，会限制一个源（域名）从其他源加载资源。

在开发 API 或 web 应用时，通常需要让不同域的客户端（如来自不同域的前端应用）能够访问你的服务。这时，`cors` 包就变得非常有用。

5.1 使用 `cors` 包的基本步骤：

1. **安装 `cors`**：在你的 Node.js 项目中，通过 npm 安装 `cors` 包：

   ```
   npm install cors
   ```

2. **在你的应用中引入并使用 `cors`**：通常，你会在你的主应用文件（如 `app.js` 或 `server.js`）中引入并使用它。例如，如果你正在使用 Express.js，代码可能如下所示：

   ```
   const express = require('express');
   const cors = require('cors');
   const app = express();

   // 使用默认的 CORS 配置
   app.use(cors());

   // ... 其他中间件和路由 ...

   app.listen(3000, () => {
     console.log('Server is running on port 3000');
   });
   ```

   在这个例子中，`app.use(cors());` 使得所有的路由都允许跨源请求。

3. **自定义 CORS 配置**：`cors` 包允许自定义哪些源可以访问你的应用。例如，你可以只允许特定的域名访问，或者为不同的路由配置不同的 CORS 规则：

   ```
   // 只允许特定的域名访问
   const corsOptions = {
     origin: 'https://www.example.com',
   };
   app.use(cors(corsOptions));

   // 为特定路由配置不同的 CORS 规则
   app.get('/public-data', cors(), (req, res) => {
   ```

```
  res.json({ msg: 'This is public' });
});

const privateCors = cors({
  origin: 'https://private.example.com',
});
app.get('/private-data', privateCors, (req, res) => {
  res.json({ msg: 'This is private' });
});
```

4. **处理预检请求（Pre-flight）**：对于某些跨源请求，浏览器会先发送一个预检请求（HTTP OPTIONS 请求），以确认服务器允许跨源请求的具体规则。`cors` 包自动处理这些预检请求。

通过上述步骤，你可以在你的 Node.js 应用中轻松地启用和管理 CORS。这样可以确保你的 API 既安全又易于跨域访问，特别是在前后端分离的开发模式中。

14. dotenv

`dotenv` 是一个简单的 Node.js 包，它允许你从一个 `.env` 文件加载环境变量到 `process.env`。这样做的好处是可以将配置与代码分离，增加安全性（例如，防止将敏感信息如数据库密码直接硬编码在代码中），并便于在不同的环境（开发、测试、生产）中使用不同的设置。

6.1 使用 `dotenv` 包的基本步骤：

1. **安装 `dotenv`**：在你的 Node.js 项目中，通过 npm 安装 `dotenv` 包：

   ```
   npm install dotenv
   ```

2. **创建 `.env` 文件**：在你的项目根目录下创建一个 `.env` 文件。在这个文件中，你可以设置键值对来定义环境变量。例如：

   ```
   DB_HOST=localhost
   DB_USER=root
   DB_PASS=s1mpl3
   ```

   这个文件应该被添加到 `.gitignore` 或类似的版本控制排除列表中，以防止将敏感信息上传到版本控制系统。

3. **在应用程序中加载环境变量**：在你的主应用文件（如 `app.js` 或 `server.js`）的最顶部，添加以下代码来加载 `.env` 文件中的环境变量：

   ```
   require('dotenv').config();
   ```

   通过这样做，`dotenv` 包会将 `.env` 文件中的键值对加载到 `process.env` 对象中。这意味着，之后你可以通过 `process.env.DB_HOST`、`process.env.DB_USER` 和 `process.env.DB_PASS` 来访问这些环境变量。

4. **使用环境变量**：现在你可以在应用中使用这些环境变量了。例如，你可以在配置数据库连接时使用它们：

```
const dbConfig = {
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  // 其他配置...
};
```

6.2 注意事项：

- **安全性**: 避免将 `.env` 文件上传到公共代码库，特别是当文件包含敏感信息时。
- **默认值和类型转换**: 有时候你可能需要为某些环境变量设置默认值，或者需要将字符串值转换为不同的类型（如数字或布尔值）。你可以在代码中手动处理这些情况。
- **不同环境的 `.env` 文件**: 对于不同的环境（开发、测试、生产），你可能需要不同的 `.env` 文件。可以通过在启动应用时设置不同的环境变量来实现，或使用像 `dotenv-flow` 这样的包来支持多个 `.env` 文件。

`dotenv` 是一种简单有效的方式，用于管理 Node.js 应用的配置。通过使用它，你可以轻松地在不同环境中切换配置，而无需更改代码。

15. express

Express 是一个灵活、轻量级的 Node.js Web 应用框架，广泛用于构建各种 Web 应用和 API。它是 Node.js 最流行的 Web 框架之一，提供了一套丰富的功能来简化 Web 和移动应用程序的开发。

7.1 主要特点

1. **中间件架构**：Express 应用是一系列中间件函数调用。每个中间件可以执行代码、更改请求和响应对象、结束请求-响应循环或调用堆栈中的下一个中间件。
2. **路由**：路由是定义应用响应客户端请求的方式，例如对不同 URL 路径和 HTTP 请求方法的响应。Express 路由提供了构建单页、多页和混合 Web 应用的灵活性。
3. **简单性和灵活性**：Express 不强迫你遵循严格的规则或结构，你可以根据项目需求自由地组织你的代码和文件结构。
4. **模板引擎支持**：Express 支持多种模板引擎，如 Pug、Mustache、EJS 等，以动态渲染 HTML 页面。
5. **错误处理**：Express 提供了一个内置的错误处理机制，并允许自定义错误处理逻辑。
6. **集成数据库支持**：可以与各种数据库无缝集成，如 MongoDB、MySQL、PostgreSQL 等。

7.2 快速入门示例

以下是一个简单的 Express 应用示例：

```
const express = require('express');
const app = express();

// 定义一个路由
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// 启动服务器
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

7.3 安装

要开始使用 Express，你首先需要通过 npm 安装它：

```
npm install express
```

7.4 使用场景

Express 常用于构建以下类型的应用：

- 单页应用（SPA）
- 多页应用（MPA）
- REST API
- 各种 Web 服务

7.5 总结

Express 以其简单性、灵活性和强大的功能集，在 Node.js 社区中获得了广泛的认可和使用。它使得开发 Web 应用和 API 变得简单快捷，无论是小型项目还是大型企业级应用。

16. log4js

`log4js` 是一个在 Node.js 环境下使用的日志记录库，它提供了丰富的日志记录功能，帮助开发者在应用程序中有效地记录和管理日志信息。这个库是受 Java 中流行的 `log4j` 库启发而开发的。

8.1 主要特点

1. **多种日志级别**：`log4js` 支持不同的日志级别，如 `TRACE`、`DEBUG`、`INFO`、`WARN`、`ERROR`、`FATAL` 等，允许你根据需要记录不同严重性级别的信息。
2. **多种日志输出方式**：支持将日志输出到控制台、文件、远程服务器等多种媒介。
3. **日志分类**：可以创建不同的日志分类（或称为“日志记录器”），每个分类可以有自己的日志级别和输出目的地。
4. **灵活的配置**：`log4js` 提供了灵活的配置选项，可以在 JSON 文件或代码中配置。
5. **日志格式化**：支持自定义日志的格式。

8.2 使用示例

以下是一个基本的 `log4js` 使用示例：

```
const log4js = require('log4js');

// 配置 log4js
log4js.configure({
  appenders: { 'out': { type: 'stdout' }, 'app': { type: 'file', filename:
'application.log' } },
  categories: { default: { appenders: ['out', 'app'], level: 'debug' } }
});

// 创建一个记录器
const logger = log4js.getLogger();

// 记录不同级别的日志
logger.trace('This is a Trace log');
logger.debug('This is a Debug log');
logger.info('This is an Info log');
logger.warn('This is a Warn log');
logger.error('This is an Error log');
logger.fatal('This is a Fatal log');
```

在此示例中，我们配置了两个日志输出器（appender）：一个输出到控制台，另一个输出到文件。我们还设置了默认的日志级别为 `debug`，这意味着所有 `debug` 级别及以上的日志都会被记录。

8.3 安装

你可以通过 npm 安装 `log4js`：

```
npm install log4js
```

8.4 使用场景

`log4js` 适用于需要详细日志记录的各种 Node.js 应用程序，如 Web 服务器、后台服务、命令行工具等。它对于故障排查、监控和分析应用程序的行为非常有帮助。

8.5 总结

总的来说，`log4js` 是一个功能强大且灵活的日志记录工具，适用于各种级别的 Node.js 项目。它提供了详细的日志记录能力，帮助开发者有效地追踪和分析应用程序的运行情况。

17. mongoose

Mongoose 是一个 MongoDB 对象建模工具，设计用于在异步环境下工作。它为 MongoDB 提供了直观、基于模式（Schema）的解决方案来建模应用程序数据。它还包括内置类型转换、验证、查询构建、业务逻辑钩子（hooks）等功能。

9.1 基本概念

1. **模式（Schema）**：Mongoose 的核心概念之一是模式（Schema）。模式允许你定义文档的结构和属性（字段），以及它们的类型、验证要求等。
2. **模型（Model）**：模型是基于模式定义的构造函数。你可以使用它来创建、查询、更新、删除文档等。
3. **文档（Document）**：文档是模型的实例，它们具有数据库记录的行为和属性。文档对应 MongoDB 中的一个文档。

9.2 使用 Mongoose 的步骤

1. **安装 Mongoose**: 首先，在你的 Node.js 项目中安装 Mongoose。

```
npm install mongoose
```

2. **连接到 MongoDB**: 使用 Mongoose 连接到你的 MongoDB 数据库。

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my_database', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

3. **定义模式（Schema）**：定义一个模式来表示你的文档结构。

```
const Schema = mongoose.Schema;

const blogSchema = new Schema({
  title: String,
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});
```

4. **创建模型（Model）**：基于模式创建一个模型。

```
const Blog = mongoose.model('Blog', blogSchema);
```

5. **使用模型**: 使用模型来创建、查询、更新或删除文档。

```
// 创建一个文档
const myBlog = new Blog({
  title: 'Introduction to Mongoose',
  author: 'John Doe',
  body: 'Mongoose makes using MongoDB easy.'
});

// 保存到数据库
myBlog.save((err) => {
  if (err) return handleError(err);
  // 保存成功的操作
});

// 查询文档
Blog.find({ author: 'John Doe' }, (err, blogs) => {
  if (err) return handleError(err);
  // 找到的文档处理
```

```
});
```

9.3 高级功能

- **验证（Validation）**：Mongoose 提供了强大的验证功能。你可以在模式中定义验证逻辑。
- **中间件（Middleware）**：Mongoose 允许你定义在操作执行前后运行的中间件（也称为预/后置钩子），这对于业务逻辑非常有用。
- **虚拟属性（Virtuals）**：虚拟属性是你可以在模式中定义的属性，它们不会持久化到 MongoDB。它们适用于计算生成的字段。
- **静态和实例方法**：你可以为模型和文档定义自己的静态方法和实例方法。

Mongoose 是 MongoDB 在 Node.js 中的强大工具，它极大地简化了数据的操作和管理工作。通过使用模式和模型，你可以轻松地为你的数据定义结构，确保数据完整性，并利用 MongoDB 强大的功能。

18. morgan

Morgan 是一个用于 Node.js 的 HTTP 请求日志中间件，广泛应用于使用 Express.js 框架的 Web 应用程序中。它可以自动记录请求信息，如请求方法、URL、响应时间、响应状态等，有助于监控和调试应用程序。

10.1 基本特性

1. **日志格式**：Morgan 提供了多种预设的日志格式，如 `combined`、`common`、`dev`、`short`、`tiny` 等，每种格式记录的信息有所不同。
2. **自定义日志格式**：除了预设格式外，Morgan 还允许开发者自定义日志格式。
3. **日志目的地**：日志可以输出到控制台或写入文件，或通过流的方式输出到其他地方，如日志管理系统。
4. **简单集成**：与 Express.js 等框架轻松集成，只需几行代码即可启用。

10.2 使用 Morgan 的步骤

1. **安装 Morgan**：在你的 Node.js 项目中安装 Morgan。

   ```
   npm install morgan
   ```

2. **引入并使用 Morgan**：在你的主应用文件（如 `app.js` 或 `server.js`）中引入 Morgan，并设置为中间件。

   ```
   const express = require('express');
   const morgan = require('morgan');
   const app = express();

   // 使用预设的日志格式
   app.use(morgan('dev'));

   // ... 其他中间件和路由 ...

   app.listen(3000, () => {
     console.log('Server is running on port 3000');
   });
   ```

   在这个例子中，`morgan('dev')` 使用了 `dev` 格式，它会输出简洁的错误调试信息。

3. **自定义日志格式**: 如果预设格式不符合你的需求，可以自定义日志格式。

```
app.use(morgan(':method :url :status :res[content-length] - :response-time
ms'));
```

这将输出 HTTP 方法、URL、响应状态、响应内容长度和响应时间。

4. **日志写入文件**: 如果需要将日志写入文件，可以使用 Node.js 的文件系统模块。

```
const fs = require('fs');
const path = require('path');
const accessLogStream = fs.createWriteStream(path.join(__dirname,
'access.log'), { flags: 'a' });

app.use(morgan('combined', { stream: accessLogStream }));
```

10.3 注意事项

- **性能考虑**: 在生产环境中记录大量日志可能会影响性能。建议根据需求调整日志级别。
- **安全和隐私**: 谨慎记录敏感信息，避免将用户敏感数据写入日志。
- **日志管理**: 对于大型应用，考虑使用专业的日志管理系统，以便于日志的查询、监控和分析。

Morgan 是一个强大而灵活的工具，能够帮助开发者获取请求级别的洞察，从而更好地了解和优化他们的应用程序。

19. swagger-jsdoc

`swagger-jsdoc` 是一个用于 Node.js 应用的库，它结合了 Swagger（现在称为 OpenAPI）的功能，允许开发者通过 JSDoc 注释直接在代码中定义 API 文档。这样，你可以在编写代码的同时编写 API 文档，再通过 `swagger-jsdoc` 生成符合 OpenAPI 规范的 API 文档。

11.1 主要特点

1. **代码中的文档**：通过在代码中添加特定格式的注释，可以直接从源代码生成 API 文档。
2. **符合 OpenAPI 规范**：生成的文档符合 OpenAPI（原Swagger）规范，这是一种广泛使用的 API 描述语言。
3. **易于集成**：可以轻松地与 Express、Koa 或其他 Node.js Web 框架结合使用。
4. **实时更新**：当你更新了代码中的注释后，文档也会相应更新，保持同步。
5. **支持多种输出格式**：生成的 API 文档可以是 JSON 或 YAML 格式，可以被 Swagger UI 或其他兼容 OpenAPI 的工具使用。

11.2 使用示例

首先，安装 `swagger-jsdoc`：

```
npm install swagger-jsdoc
```

在代码中使用：

```
const swaggerJSDoc = require('swagger-jsdoc');

// Swagger 定义
const swaggerDefinition = {
```

```
  openapi: '3.0.0',
  info: {
    title: 'Example API',
    version: '1.0.0',
    description: 'A sample API',
  },
};

// 配置选项
const options = {
  swaggerDefinition,
  // 路径到 API 文档
  apis: ['./routes/*.js'],
};

// 初始化 swagger-jsdoc
const swaggerSpec = swaggerJSDoc(options);
```

然后，在你的路由文件中添加 Swagger 注释：

```
/**
 * @swagger
 * /api/path:
 *   get:
 *     summary: Retrieve something
 *     description: Detailed description here
 *     responses:
 *       200:
 *         description: Successful response
 */
```

最后，可以使用生成的 `swaggerSpec` 对象与 Swagger UI 或其他工具集成，以展示和测试 API。

11.3 使用场景

`swagger-jsdoc` 非常适合用于需要自动生成和维护 API 文档的项目。对于快速发展的项目，手动更新独立的 API 文档可能既费时又易出错。通过使用 `swagger-jsdoc`，开发者可以确保文档始终与代码保持一致，并且易于维护。它适用于小型到大型的各种 Web 服务和 RESTful API 项目。

20. swagger-ui-express

`swagger-ui-express` 是一个用于在 Express 应用程序中集成 Swagger UI 的中间件。Swagger（现在被称为 OpenAPI）是一个规范和完整的框架，用于描述、生产、消费和可视化 RESTful Web 服务。集成 `swagger-ui-express` 可以让你的 Express 应用具有交互式的 API 文档页面，使得 API 的测试和文档查看变得更加容易。

12.1 关键特性

- **自动生成 API 文档**: 根据 OpenAPI（Swagger）规范自动生成 API 文档。
- **交互式 UI**: 提供一个交互式的 Web 用户界面，让用户可以查看和测试 API。
- **集成到 Express 应用中**: 作为一个中间件集成到现有的 Express 应用中。

12.2 使用 `swagger-ui-express` 的基本步骤

1. **安装 `swagger-ui-express`**：在你的项目中通过 npm 安装 `swagger-ui-express` 和 `swagger-jsdoc`（用于生成 Swagger 规范的文档）。

```
npm install swagger-ui-express swagger-jsdoc
```

2. **创建 Swagger 规范文档**：使用 `swagger-jsdoc` 定义你的 API。通常，这涉及到编写 JSDoc 注释。

```
const swaggerJsdoc = require('swagger-jsdoc');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Express API with Swagger',
      version: '1.0.0',
    },
  },
  apis: ['./routes.js'], // 文件路径匹配你的路由文件
};

const swaggerSpec = swaggerJsdoc(options);
```

3. **在 Express 应用中使用中间件**：将 Swagger UI 作为中间件添加到你的 Express 应用中。

```
const express = require('express');
const swaggerUi = require('swagger-ui-express');

const app = express();

app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));

// ... 其他路由和中间件 ...

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

   在这里，`/api-docs` 路径被设置为 Swagger 文档的入口点。

4. **访问 Swagger UI**：启动你的 Express 应用后，通过访问 `http://localhost:3000/api-docs`（假设你的服务器运行在端口 3000 上）来查看和交互你的 API 文档。

12.3 注意事项

- **安全性**：由于 Swagger UI 公开了你的 API 细节和端点，你可能需要考虑在生产环境中限制对其的访问。
- **文档的准确性**：确保你的 Swagger 文档准确反映了你的 API。任何更改或更新都应该同步到 Swagger 配置中。
- **API 测试**：Swagger UI 提供了测试 API 端点的功能。这对于调试和验证 API 行为非常有用。

集成 `swagger-ui-express` 是提高 API 可用性和可维护性的有效方式，它为开发者和最终用户提供了一个强大的界面来理解和使用你的 API。

21.ngrok

`ngrok` 是一个反向代理工具，它可以将本地运行的服务器暴露到公共的互联网上。这个工具非常有用，特别是在开发阶段，当你需要分享你的应用或测试Webhook等功能时。

13.1 主要特点

1. **本地到公网映射**：`ngrok` 通过创建一个从公网到本地机器的安全隧道，让外部可以访问本地运行的服务。
2. **无需配置**：不需要进行复杂的网络配置，如设置 NAT 或防火墙规则。
3. **即时在线**：启动 `ngrok` 后，立即获得一个公网访问地址。
4. **支持多种协议**：支持 HTTP、HTTPS 和 TCP 协议。
5. **内置的网络分析工具**：提供请求监控和分析，方便调试和检查 HTTP 请求。
6. **安全连接**：所有通过 `ngrok` 的连接都是加密的，保证数据安全。

13.2 使用示例

1. **下载并安装**：首先从 ngrok 官网 下载并解压 `ngrok`。
2. **启动本地服务**：假设你在本地运行了一个 Web 服务，监听在 8080 端口。
3. **启动 ngrok**：在命令行中运行以下命令来启动 `ngrok`，并将其映射到你的本地端口：

   ```
   ./ngrok http 8080
   ```

   这将在 `ngrok` 服务器上分配一个唯一的 URL，并将所有流量转发到本地的 8080 端口。
4. **访问公网 URL**：`ngrok` 提供的公网 URL 可以在任何地方访问，从而直接连接到你的本地服务。

13.3 使用场景

- **Web 开发和测试**：在开发过程中，如果你需要与外部服务（如第三方API、Webhook）交互，`ngrok` 可以使这些服务能够直接与你的本地开发服务器通信。
- **演示和分享**：如果你想向客户或同事展示在本地开发的应用，使用 `ngrok` 可以立即实现。
- **移动应用开发**：当开发移动应用需要连接到运行在本地计算机上的后端服务时，`ngrok` 提供了一个简便的解决方案。

13.4 关闭进程
CMD

```
tasklist | findstr "ngrok"
taskkill /F /PID 1234
```

PowerShell

```
Get-Process ngrok
Stop-Process -Id 1234 -Force
```

13.5 注意事项

- `ngrok` 提供了免费和付费版本，免费版本有一些限制，例如隧道的生命周期和每月的流量限制。
- 使用公共 URL 暴露本地服务时要注意安全风险，尤其是当暴露敏感或未加密的数据时。

22.@swagger

```
/**
 * @swagger
 * /api/products:
 *   post:
 *     tags:
 *       - Product
 *     summary: 创建新产品
 *     description: 添加一个新产品到产品列表。
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               name:
 *                 type: string
 *               price:
 *                 type: number
 *     responses:
 *       201:
 *         description: 创建成功，返回新创建的产品
 *       400:
 *         description: 创建失败
 */
```

这段代码是一个 Swagger 注释，用于生成关于特定 API 路径（在这个例子中是 `/api/products`）的文档。Swagger 注释是用于描述和定义 API 接口的详细信息，它们通常放置在实际的 API 实现代码附近。这个注释包含了以下几个部分：

1. **路径和方法 (`/api/products: post`)**:
   - `@swagger`: 这个标记表示接下来的注释是 Swagger 文档的一部分。
   - `/api/products`: 表示这个文档是关于 `/api/products` 路径的。
   - `post`: 指的是 HTTP 请求方法，这里是 POST，意味着这是一个创建新资源（在这个例子中是产品）的接口。
2. **标签 (`tags`)**:
   - `Product`: 这个标签是用来分类和组织 API 文档的。在这个例子中，这个 POST 请求被归类为 `Product` 类别。
3. **摘要和描述 (`summary` 和 `description`)**:
   - `summary`: 提供了这个 API 接口的简短概述，这里是 "创建新产品"。
   - `description`: 给出了关于这个 API 接口更详细的信息，这里说明了这个接口是用来 "添加一个新产品到产品列表"。
4. **请求体 (`requestBody`)**:
   - `required: true`: 表明请求体是必需的。
   - `content`: 描述了请求体的内容类型和结构。这里是 `application/json` 类型，包含 `name`（字符串类型）和 `price`（数字类型）两个属性。

5. **响应 (`responses`)**:
    - 定义了不同响应状态码及其描述。这里定义了两种响应：
        - `201`：创建成功时的响应，描述为 "创建成功，返回新创建的产品"。
        - `400`：创建失败时的响应，描述为 "创建失败"。

这些信息被 Swagger 工具用来自动生成对应的 API 文档，包括请求和响应的格式、路径、可能的状态码等。这对于 API 的测试、文档化和其他开发者的理解非常有帮助。