————————————————

版权声明：本文为CSDN博主「男星」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/include\_he/article/details/122700113

# OpenSSL 3.0设计介绍

## 介绍

本文档概述了OpenSSL 3.0的设计，这是OpenSSL在1.1.1之后的下一个版本。它假设熟悉名为OpenSSL Strategic Architecture的文档，并具有OpenSSL 1.1.x的工作知识。

OpenSSL 3.0版本对绝大多数现有应用程序的影响最小。几乎所有表现良好的应用程序只需要重新编译即可。

OpenSSL 3.0中的大多数更改都是内部架构重组，以促进长期可支持的加密框架，从而更好地将算法实现与算法API分离。这些结构更改还支持更易于维护的 OpenSSL FIPS 加密模块 3.0。

在 OpenSSL 3.0 中，不会删除当前标记的已弃用 API。

在 OpenSSL 3.0 中，许多其他低级函数将被标记为已弃用的 API。

OpenSSL 3.0 将支持同时处于 FIPS 模式（使用 OpenSSL FIPS 加密模块 3.0）的 TLS 连接和同时处于非 FIPS 模式的 TLS 连接的应用程序。

**本文中使用的术语**

算法(Algorithm)，或有时是加密算法(cryptographic algorithm)，是一种执行一组操作（如加密或解密）的方法。我们对该术语的使用是抽象的，通常通过其名称表示算法，例如"aes-128-cbc"。

算法实现(Algorithm implementation)，或者有时只是实现，是算法的具体实现。这主要以一组函数的形式在代码中表示。

CAVS是加密算法验证系统。用于测试加密实现是否符合 FIPS 标准的工具。

CMVP是加密模块验证程序。验证加密实现是否符合 FIPS 标准的过程。

EVP是由 libcrypto 实现的一系列 API，它使应用程序能够执行加密操作。EVP API 的实现使用核心组件(Core)和提供者(Provider)组件。

核心(Core)，是libcrypto中的一个组件，它使应用程序能够访问提供者(Provider)提供的算法实现。

CSP是关键安全参数。这包括任何信息（例如私钥、密码、PIN码等），这些信息可能会在未经授权的披露或修改时危及模块的安全性。

显式提取(Explicit Fetch)，是一种用于查找算法实现的方法，其中应用程序显式调用以查找实现并提供搜索条件。

FIPS是联邦信息处理标准。这是美国政府制定的一套标准。特别是，FIPS 标准 140-2 适用于加密软件。

FIPS 模块是加密算法的实现，已由 CMVP 验证为符合 FIPS 标准。在 OpenSSL 中，FIPS 模块作为提供者(Provider)实现，并以动态可加载的模块形式提供。

隐式提取(Implicit Fetch)，是一种用于查找算法实现的方法，其中应用程序不会显式调用来查找实现，因此使用默认搜索条件。

完整性检查(Integrity Check)，是在加载 FIPS 模块时自动运行的测试。模块会对自身进行校验和，并验证它是否没有被无意中更改。

KAS是关键协议计划。两个通信方通过该方法可以在它们之间约定共享密钥。

KAT是已知答案测试。用于执行 FIPS 模块运行状况检查的一组测试。

libcrypto是由OpenSSL实现的共享库，它为应用程序提供了对各种加密相关功能的访问。

libssl是由OpenSSL实现的共享库，它为应用程序提供了作为客户端或服务器创建SSL / TLS连接的能力。

库上下文(Library Context)，是保存库"全局"数据的不透明结构。

操作(Operation)，是对数据执行的一类功能，例如计算摘要，加密，解密等。算法可以提供一个或多个操作。例如，RSA 提供非对称加密、非对称解密、签名、验签等

参数(Parameters)，是一组与实现无关的键值对，用于在核心(Core)和提供者(Provider)之间传递对象数据。例如，它们可用于传输私钥数据。

POST是指在安装时、上电时（即每次为应用程序加载 FIPS 模块时）或按需运行的 FIPS 模块上电自检（也称为开机自检）。这些测试包括完整性检查和 KAT。如果KAT在安装时成功运行，则无需在通电时再次运行，但始终执行完整性检查。

属性(Properties)，提供者(Provider)使用属性来描述其算法实现的功能。它们还用于应用程序查询以查找特定实现。

提供者(Provider)，是提供一个或多个算法实现的单元。

提供者模块(Provider module)，是动态可加载模块形式的提供者。

## 结构

该结构具有以下特点：

1、基本服务构成了应用程序和提供者(Provider)可用的构建块。(例如BIO、X 509、SECMEM、ASN.1等)。

2、提供者(Provider)，实现加密算法和支持服务。一个算法可以由多个操作组成(例如RSA可能有“加密”、“解密”、“签名”、“验签”等)。类似地，一个操作(例如“签名”)可以由多个算法(如RSA和ECDSA)实现。提供者(Provider)具有用于算法的加密原语的实现。这一版本将包括下列提供者(Provider)：

Default Provider，默认的提供者(Provider)，包含了没有被废弃的OpenSSL密码算法，这将是内置的(即libcrypto的一部分)

Legacy Provider，废弃算法的提供者(Provider)，包含已经被废弃算法的实现(例如，DES、MCC 2、MD2、Blowfish、CAST)

Engines Provider，使用引擎的提供者(Provider)，包括为较旧版本的OpenSSL设计的引擎与新的基于提供者(Provider)的方法之间的兼容层。

FIPS Provider，它实现了OpenSSLFIPS密码模块3.0；它可以在运行时动态加载。

3、核心(Core)，允许访问提供者(Provider)为应用程序(和其他提供者(Provider))提供的操作。核心是定位具体操作实现的机制。

4、协议实现。例如TLS，DTLS

本文档有许多对"EVP API"的引用。这是指"应用程序级"操作，例如公钥签名、生成摘要等。这些函数包括EVP\_DigestSign、EVP\_Digest等。EVP API 还封装了用于执行这些服务的加密对象，如EVP\_MAC\_init、EVP\_PKEY、EVP\_CIPHER、EVP\_MD、EVP\_MAC等。提供者(Provider)为后一组实现后端功能。这些对象的实例可以隐式或显式绑定到提供者(Provider)，具体取决于应用程序的需要。这将在下面的提供者(Provider)设计中更详细地讨论。

该体系结构具有以下特点：

1、EVP层是在提供者(Provider)中实现的操作的外层封装。大多数调用都是直接通过的，几乎没有预处理或后处理。

2、提供新的EVP API影响核心(Core)如何选择(或查找)用于任何给定EVP调用的操作的实现。

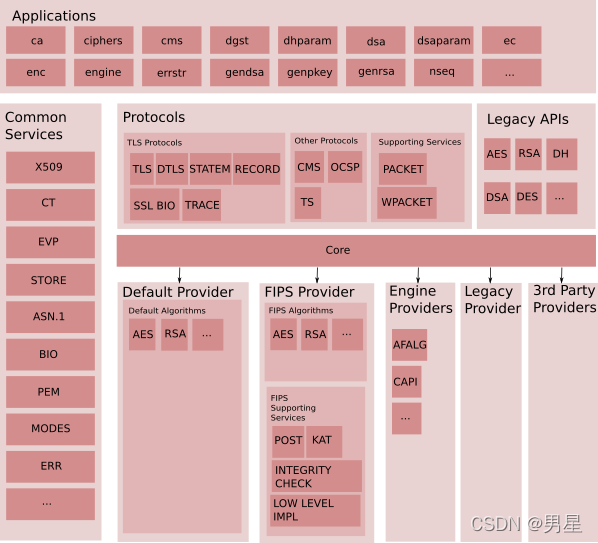
3、信息将以不可知的方式在libcrypto和提供者(Provider)之间传递。

4、遗留API(例如，不通过EVP层的低级加密API)将被废弃。请注意，对于没有被废弃5、算法有遗留API(例如，AES不是废弃算法，而是AES\_encrypt是一个遗留API)。

6、OpenSSL FIPS 加密模块将作为动态加载的提供者(Provider)实现。它将是独立的(即只能依赖于核心(Core)提供的系统运行时库和服务)。

### 概念组件视图

OpenSSL体系结构中概念组件的概述如下图所示。请注意，图中组件的存在并不表示该组件是公共API或用于最终用户的直接访问或使用。



新组件(在以前的体系结构中不存在)如下：

1、核心(Core)：这是一个基本组件，它将操作请求(例如加密)连接到该操作的提供者(Provider)。它提供了定位提供指定操作的算法的实现的能力，给出了实现必须实现的一组属性。例如，加密算法的属性可以至少包括“FIPS”。

2、默认提供者(Default Provider)：实现一组默认算法。

3、FIPS提供者(FIPS Provider)：实现一组经过FIPS验证并通过核心(Core)提供的算法。这包括下列支持服务：

POST：自我测试的能力

KAT： 已知答案测试

完整性检查(Integrity Check)

低级别实现：这是一组实际上实现加密原语的组件(以满足FIPS强制要求的自包含需求)。

4、引擎提供者(Engines Provider)：允许现有引擎通过核心调用时工作的中间件。

5、遗留提供者(Legacy Provider)：提供旧算法的实现，这些算法将通过EVP级别的API公开。

6、第三方提供者(3rd Party Providers)：最终，第三方可以提供他们自己的提供者(Provider)。第三方提供者(Provider)，就像任何其他提供者(Provider)一样，实现了一组算法，应用程序和其他提供者(Provider)可以通过核心(Core)访问这些算法。

### 封装视图

以上概念组件视图中描述的各种组件实际上封装为：

供用户使用的可执行应用程序(Executable application(s))

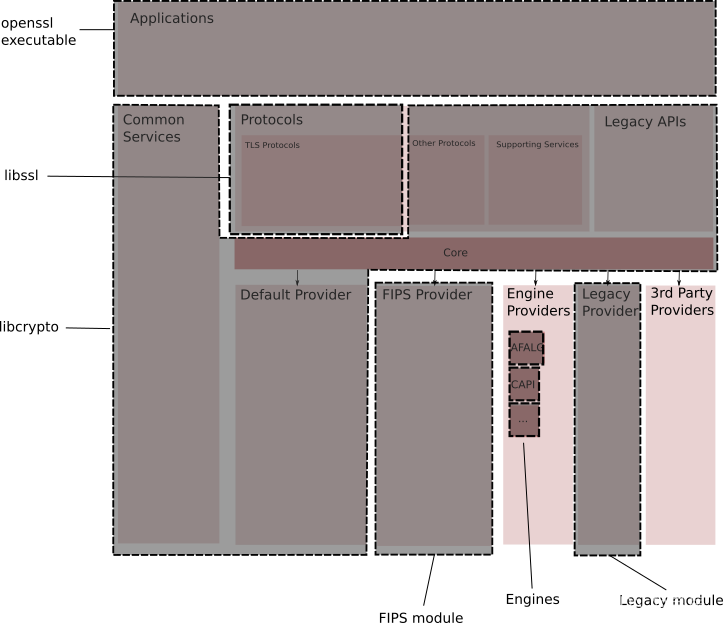
供应用程序使用的库(Libraries)

可动态加载模块(Dynamically loadable module(s))，供核心(Core)使用。

OpenSSL3.0将提供多个不同的打包选项(例如，一个库libcrypto包含除FIPS提供者(Provider)之外的所有内容，以及所有提供者(Provider)作为单独的动态可加载模块)。

哪些动态可加载模块被注册、使用或可用，将能够在运行时进行配置。

下图以实际包的形式描述了体系结构。



新发布的外部包有：

1、FIPS模块(FIPS module)。它包含FIPS Provider，它实现了一组经过FIPS验证的算法，这些算法可以通过Core使用。FIPS提供者是OpenSSL FIPS加密模块3.0。

OpenSSL不会试图阻止用户犯错误，但是会记住典型的用户使用和“安全”。默认情况下，将构建并安装FIPS提供者。

OpenSSL将能够执行安全检查，以检测用户是否以对FIPS有影响的方式从发行版中修改了源，并构建FIPS提供者(在最大努力的基础上)，除非提供了覆盖选项。

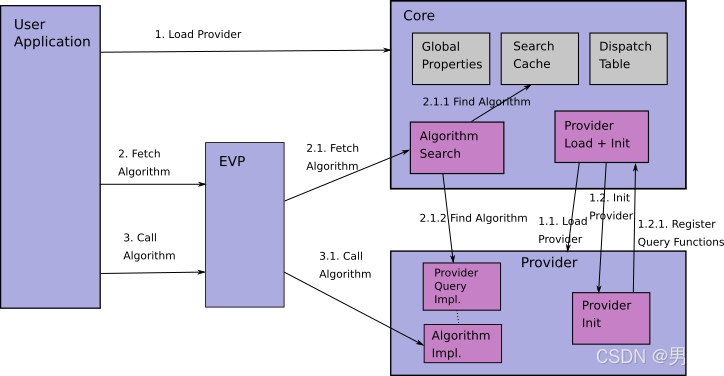
OpenSSL需要确保存在一种机制，使最终用户能够确定他们对FIPS模块的使用是否在正式验证下的允许使用范围内。

FIPS模块的版本控制将在验证时与基本OpenSSL版本号对齐。并不是所有的OpenSSL版本都需要更新FIPS模块。因此，当一个新的FIPS模块版本发布时，它的版本号可能会与之前的版本存在差距或跳跃。

2、遗留模块(Legacy module)。它包含了废弃算法的实现。

### 核心(Core)和提供者(Provider)设计

与核心(Core)和提供者(Provider)设计相关的交互如下图所示。有四个主要组件:用户应用程序、EVP组件、核心(Core)和加密提供者(Provider)。(可能有多个提供者(Provider)，但这与此无关)。



核心(Core)具有以下特点:

支持提供者(Provider)的发现、加载、初始化和卸载。

支持基于属性的算法查询。

实现了算法查询的缓存和实现细节。

在一个“库上下文(library context)”范围中操作，库上下文包含全局属性数据、搜索缓存和调度表(dispatch table)

提供者(Provider)有以下特点：

提供了对特定算法实现的访问。

将算法实现与一组定义良好的属性关联起来。

支持以与实现无关的方式传递参数。

可以在任何时间点装载。

有一个特定的模块入口点。

下面的小节描述了应用程序用于加载提供者(Provider)、获取算法实现并使用它的流程。此外，本节详细描述了如何命名算法、属性和参数；如何处理算法查询；如何注册和初始化算法；以及如何加载提供者(Provider)。

为了使应用程序能够使用一个算法，它必须首先通过一个算法查询“提取(fetch)”它的一个实现。我们的设计目标是能够支持显式(预先)的算法提取和正在使用的算法的提取。默认情况下，我们期望在使用时进行提取(例如使用EVP\_sha256())，因此算法通常在“初始化(init)”函数中获取，并绑定到上下文对象(通常命名为ctx)。显式提取选项将通过新的API调用来实现(例如，EVP\_MD\_fetch())。

上面的图表显示了显式提取方法。这些步骤如下：

1、每个提供者(Provider)都需要加载。这将隐式发生(默认提供者(Provider)或通过配置指定)，也可能由应用程序显式请求。Load Provider包括动态共享对象加载Load(视需要而定)和初始化Init。

1.1、核心(Core)从外部将模块加载到内存中(如果默认提供者(Provider)已经在内存中，则不需要)。

1.2、核心(Core)调用提供者(Provider)的入口点让提供者(Provider)初始化自己。

在入口点函数中，提供者(Provider)使用从核心(Core)传入的值初始化一些提供者(Provider)变量。如果初始化成功，则向核心(Core)返回提供者(Provider)算法实现查询回调。

2、用户应用程序通过调用提取动作来请求算法。

2.1、EVP搜索将全局属性与特定于调用的属性和算法标识相结合，以找到相应的算法实现，然后创建并返回一个库句柄。如EVP\_MD, EVP\_CIPHER的申请。

A、实现调度表的第一次搜索是在内部缓存中进行的。

B、如果第一次搜索失败，则通过询问提供者(Provider)是否具有所查询属性的所述算法的实现来进行第二次搜索。当完成此搜索时，结果数据将被缓存，除非提供者(Provider)选择退出缓存，以便在第一次搜索(2.1.1)中使用。例如，PKCS#11提供者(Provider)可能选择退出缓存，因为随着时间的推移，其算法可能变得不可用。

3、然后，用户应用程序通过EVP API使用该算法(例如，EVP\_DigestInit(), EVP\_DigestUpdate(), EVP\_DigestFinal()(等)

3.1、函数指针被调用，并在实现的提供者(Provider)中结束，以执行请求的加密算法。

对于现有的EVP\_{algorithm}()函数(如：EVP\_sha256()）基本保持不变。特别是，当EVP\_{algorithm}()调用返回，而是在上下文对象(例如，上下文对象)时隐式地发生。EVP\_MD\_CTX)在各自的EVP init函数中绑定。具体来说，步骤2.1就发生在步骤3.1之前。这被称为“隐式提取”。隐式FETCH总是在默认库上下文中操作(请参见库上下文(见下文)。

方法调度表是<function-id, function-pointer>对-其中function-id是由OpenSSL公开定义和已知的–与一组可用于标识每个特定实现的属性一起使用。核心(Core)可以使用属性查询并找到相应的调度表，用于适用的操作。这种方法允许提供者(Provider)灵活地传递对OpenSSL代码可以用来动态创建其方法结构的函数的引用。

提供者(Provider)可以在任何时间点加载。可以请求在任何时间点卸载。在卸载提供者(Provider)时，应由应用程序确保当前不使用或引用提供者(Provider)。如果尝试使用不再可用的实现，则将返回错误。

当前应用程序可以对EVP\_{algorithm}()而不是新的获取方法是：

Const指针

不需要由应用程序释放。

可以安全地进行比较，因为检查算法是相同的(特别是比较指针EVP\_CIPHER, EVP\_MD(等)

直接使用应用程序的显式提取(而不是使用现有的EVP\_{algorithm}()(函数)语义将是不同的：

非Const指针

需要由应用程序释放。

指针不能安全地相互比较(在下一段中会有更多的介绍)

将有新的API来测试可以用于显式获取对象和静态变体对象的对象的相等性–这些API只允许比较算法标识，也可以比较特定的算法实现。

### 库上下文(Library Context)

库上下文(Library Context)是保存库“全局”数据的不透明结构。OpenSSL将提供这样的结构，仅限于核心(Core)必须保存的全局数据。未来的扩张可能会包括其他现有的全局数据。应用程序将能够创建和销毁一个或多个库上下文(Library Context)，其中所有后续与核心(Core)的交互都在其中运行。如果应用程序不创建并提供自己的库上下文(Library Context)，则将使用内部默认上下文。

OPENSSL\_CTX \*OPENSSL\_CTX\_new();

void OPENSSL\_CTX\_free(OPENSSL\_CTX \*ctx);

库上下文(Library Context)可以传递给显式提取函数。如果NULL传递给它们时，将使用内部默认上下文。

可以分配多个库上下文(Library Context)，这意味着可以多次初始化任何提供者模块(Provider module)。这允许应用程序直接链接到libcrypto和它感兴趣的加载提供者(Provider)，以及独立地与使用自己的提供者模块(Provider module)的其他库链接。

### 命名

算法、参数和属性都需要名称。为了确保一致性，并使外部提供者(Provider)实现者能够以一致的方式定义新的名称，将有一个推荐或使用名称的注册中心。它将与来源分开维持。

需要为名称定义别名的能力，因为在某些上下文中，同一事物有多个名称(例如，EC曲线具有通用名称，NIST名称用于同一事物)。

### 算法实现选择的属性

算法实现(加密和非加密)将具有一些属性，这些属性将用于从可用的实现中选择实现。对于3.0，定义了两个属性：

这个实现是默认实现吗？

此实现FIPS是否已验证？

有效输入及其含义如下：



在所有情况下，属性名称都将定义为可打印的ASCII字符，并且不区分大小写。属性值可以引用，也可以不引用。未引号的值也始终是可打印的ASCII字符，并且不区分大小写。引用的值仅在原始字节比较的基础上测试是否相等。

提供者将能够提供自己的名称或值。属性定义和查询的完整语法出现在附录1-属性语法.

OpenSSL保留所有没有句点的属性名称；提供者(Provider)提供的属性名称必须在名称中包含句点。预期(但不强制)第一个期间之前的部分属性名称是或与提供者(Provider)的名称有关，通过名称空间提供一定程度的冲突避免。

### 基于属性的算法选择

算法实现选择是基于属性的。

提供者(Provider)根据它提供的算法设置属性。应用程序设置它希望在算法选择过程中将哪些属性用作筛选器–查询。

可以在以下位置指定用于获取算法实现的所需属性：

全局的，基于配置文件。

全局的，基于API调用。

在每个对象的基础上对特定的对象。例如：ssl\_ctx，ssl。

属性将在算法查找(属性值的参数规范)中使用。

属性集将以解析为每个指定属性(关键字)的属性的单个值的方式进行计算。关键字计算的优先级顺序是：

要获取的每个对象或直接指定的api参数。

API调用设置的全局(默认)属性。

在配置文件中设置的全局(默认)属性。

### 参数定义

OpenSSL核心(Core)和提供者(Provider)必须在保持OpenSSL和提供者(Provider)结构不透明的同时交换数据。所有复合值都将作为项目数组传递，使用附录2中定义的公共数据结构—OpenSSL参数传递。参数将使用它们的名称(作为字符串)来标识，每个参数都包含自己的类型和大小信息。

核心(Core)将定义一个API，将参数值数组或对值的请求传递给提供者(Provider)或特定的算法实现，而对于后者，则传递由该实现处理的关联对象。在基本机器类型的情况下，可以开

发宏来帮助构造和提取值。

### 操作和操作功能定义

算法和参数名称本质上是由提供者(Provider)控制和分配的，而libcrypto将要调用的操作和相关函数基本上是由核心(Core)控制和分配的。

对于只受核心(Core)控制的事物，我们将使用宏来命名它们，将数字作为值作为索引。分配将是递增的，即对于任何新的操作或功能，将选择下一个可用的号码。

### 算法查询

每种算法类型(例如：EVP\_MD, EVP\_CIPHER具有“提取(fetch)”功能(例如：EVP\_MD\_fetch(), EVP\_CIPHER\_fetch())。算法实现是使用它们的名称和属性来标识的。

每个提取函数将使用核心(Core)提供的服务来找到适当的实现，如核心(Core)和提供者(Provider)设计。如果已找到适当的实现，则将其构造为适当的算法结构(例如：EVP\_MD, EVP\_CIPHER)并返回到调用应用程序。

如果多个实现与传递的名称和属性相同，那么其中的一个将在检索时返回，但确切地说是没有定义的。此外，也不能保证每次都会返回相同的匹配。

### 算法查询缓存

算法查询将与其结果一起缓存。

可以刷新算法查询缓存以删除：

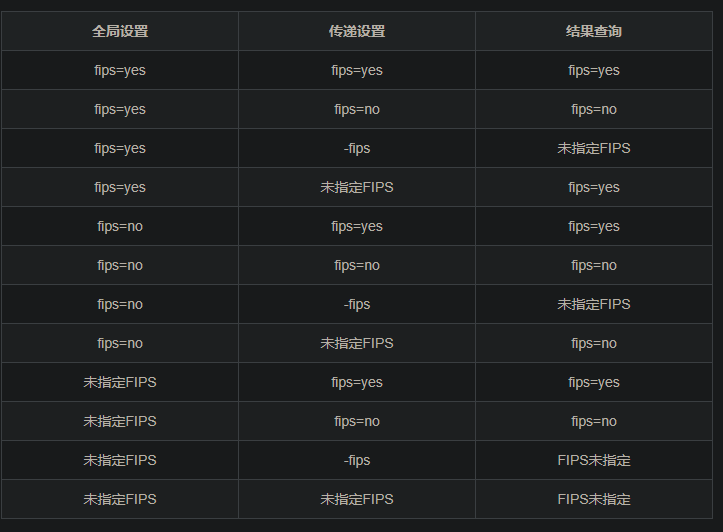
返回特定算法实现的所有查询

来自特定提供者的所有算法实现

所有算法实现

### 多级查询

为了处理全局属性和传递给特定调用(例如获取调用)的属性，全局属性查询设置将与传递的属性设置合并，除非存在冲突，特别是：

**提供者模块加载**

提供者(Provider)既可以内置，也可以动态加载模块。

所有算法都由提供者(Provider)实现。OpenSSL核心(Core)最初将没有加载提供者(Provider)，因此没有可用的算法。需要发现并加载提供者(Provider)。随后，核心(Core)可以查询它们中包含的算法实现，这些查询可能会被缓存。

如果在第一次提取时没有加载提供者(Provider)(隐式和显式)，则内置的默认提供者(Provider)将自动加载。

请注意，提供者(Provider)可能是针对较旧版本的CoreAPI编写的，而不是libcrypto中的当前版本。例如，用户必须能够运行与主要OpenSSL版本不同的FIPS提供者(Provider)模块版本。这意味着CoreAPI必须保持稳定和向后兼容(就像任何其他公共API一样)。

作为OpenSSL构建的一部分提供的所有命令行应用程序都将获得一个-provider xxx选项，该选项加载提供者(Provider)。可以在命令行上多次指定此选项(因为可以始终加载多个提供者(Provider))，如果提供者(Provider)在任何特定操作中仍未使用(例如，加载仅在执行SHA 256摘要时提供AES的提供者(Provider))，则不会出现错误。

### 查找和加载动态提供者(Provider)模块

动态提供者模块是.so，UNIX类型操作系统上的文件，或.dll，Windows类型操作系统上的文件，或其他操作系统上对应的任何文件。默认情况下，它们将安装在一个特定的目录中。

提供者(Provider)模块加载可以通过以下几种方式进行：

按需，应用程序必须确切地指定应该加载哪些提供者(Provider)模块。

按配置，将在配置文件中指定要加载的提供者(Provider)模块集。

其中一些方法可以结合起来。

提供者模块可以由完整路径指定，因此即使它不在已知的目录中，也可以加载。

在核心(Core)加载了一个提供者(Provider)模块之后，它会调用提供者模块入口点。

### 提供者模块入口点

提供者(Provider)模块必有以下特定的入口点：

int OSSL\_provider\_init(const OSSL\_PROVIDER \*provider,

const OSSL\_DISPATCH \*in,

const OSSL\_DISPATCH \*\*out

void \*\*provider\_ctx);

如果入口点不存在于动态加载的对象中，那么它就不是一个有效的模块，加载它将失败。

in是核心(Core)传递给提供者(Provider)的函数数组。

out提供者(Provider)传递给Core的提供者(Provider)函数数组。

provider\_ctx(可缩短为provctx在本文档的其他地方)是一个可选的对象，由提供者(Provider)自行创建(存储需要安全保存的数据)。此指针将被传递回适当的提供者(Provider)函数。

provider是属于核心(Core)的提供者(Provider)对象的句柄。这可以作为一个唯一的提供者标识，这在某些API调用中可能是必需的。此对象还将填充各种数据，如模块路径、提供者(Provider)的NCONF配置结构(请参阅COF/NCONF值作为参数下面是关于如何实现的想法)，然后由提供者(Provider)使用核心(Core)提供的Params getter回调来检索这些不同的值。类型OSSL\_PROVIDER是不透明的。

OSSL\_DISPATCH是实现< function-id, function-pointer >导言中提到的核心(Core)和提供者(Provider)设计:

typedef struct ossl\_dispatch\_st {

int function\_id;

void \*(\*function)();

} OSSL\_DISPATCH;

这个funcion\_id标识特定的函数，以及function是指向该函数的指针。其中的数组以function\_id设定为零。

提供者模块(Provider module)可能被链接到libcrypto，也可能不会链接到libcrypto。如果不是，那么它将无法直接访问任何libcrypto函数。所有返回到libcrypto的基本通信都将通过核心(Core)提供的回调函数进行。重要的是，由特定提供者分配的内存由相同的提供者释放。同样，libcrypto中分配的内存应该由libcrypto释放。

API将指定一组特定的回调函数编号。可以根据需要在以后的版本中添加更多的函数号，而不会破坏向后兼容性。

/\* Functions provided by the Core to the provider \*/

//由核心提供给提供者的函数

#define OSSL\_FUNC\_ERR\_PUT\_ERROR 1

#define OSSL\_FUNC\_GET\_PARAMS 2

/\* Functions provided by the provider to the Core \*/

//由提供者(Provider)提供给核心的函数

#define OSSL\_FUNC\_PROVIDER\_QUERY\_OPERATION 3

#define OSSL\_FUNC\_PROVIDER\_TEARDOWN 4

核心(Core)将设置一个特定的回调函数数组：

static OSSL\_DISPATCH core\_callbacks[] = {

{ OSSL\_FUNC\_ERR\_PUT\_ERROR, ERR\_put\_error },

/\* int ossl\_get\_params(OSSL\_PROVIDER \*prov, OSSL\_PARAM params[]); \*/

{ OSSL\_FUNC\_GET\_PARAMS, ossl\_get\_params, }

/\* ... and more \*/

};

这只是核心(Core)可能认为适合传递给提供者(Provider)的几个功能。我们也可以传递功能，以帮助日志，测试，仪器仪表等方面的需要。

一旦加载了模块并找到了特定的入口点，核心就可以调用init入口点：

/\*

\* NOTE: this code is meant as a simple demonstration of what could happen

\* in the core. This is an area where the OSSL\_PROVIDER type is not opaque.

\* 注释：注意:这段代码只是简单地演示了内核中可能发生的情况。在这个区域中，OSSL\_PROVIDER类型是不透明的。

\*/

OSSL\_PROVIDER \*provider = OSSL\_PROVIDER\_new();

const OSSL\_DISPATCH \*provider\_callbacks;

/\*

\* The following are diverse parameters that the provider can get the values

\* of with ossl\_get\_params.

\* 注释：下面是provider可以使用ossl\_get\_params获取的各种参数的值。

\*/

/\* reference to the loaded module, or NULL if built in

\* 注释：引用所加载的模块，如果内置则为NULL

\*/

provider->module = dso;

/\* reference to the path of the loaded module

\* 注释：引用已加载模块的路径

\*/

provider->module\_path = dso\_path;

/\* reference to the NCONF structure used for this provider

\* 注释：引用此提供者(Provider)使用的NCONF结构

\*/

provider->conf\_module = conf\_module;

if (!OSSL\_provider\_init(provider, core\_callbacks, &provider\_callbacks))

goto err;

/\* populate |provider| with functions passed by the provider

\* 注释：用provider传递的函数填充|provider|

\*/

while (provider\_callbacks->func\_num > 0) {

switch (provider\_callbacks->func\_num) {

case OSSL\_FUNC\_PROVIDER\_QUERY\_OPERATION:

provider->query\_operation = provider\_callbacks->func;

break;

case OSSL\_FUNC\_PROVIDER\_TEARDOWN:

provider->teardown = provider\_callbacks->func;

break;

}

provider\_callbacks++;

}

这个OSSL\_provider\_init入口点不注册所需的任何算法，但它将至少返回这两个回调以启用此过程：

1、OSSL\_FUNC\_QUERY\_OPERATION，用于查找操作的哪些实现可用。这必须返回一个OSSL\_ALGORITHM(请进一步了解)，它将算法名称和属性定义字符串映射到实现调度表。此函数还必须能够指示结果数组是否可以由Core缓存。下文将对此作进一步详细解释。

2、OSSL\_FUNC\_TEARDOWN，在卸载提供者(Provider)时使用。

在OSSL\_provider\_init()调用成功后提供者(Provider)才可以进行注册回调函数。

### 提供者初始化和算法注册

算法提供了一组操作(功能、特性等)。这些操作是通过函数调用的。例如，RSA算法提供签名和加密(两个操作)，它们通过init, update, final签名和init, update, final用于加密的函数。函数集由上层EVP代码的实现决定。

操作由唯一的数字标识。例如：

#define OSSL\_OP\_DIGEST 1

#define OSSL\_OP\_SYM\_ENCRYPT 2

#define OSSL\_OP\_SEAL 3

#define OSSL\_OP\_DIGEST\_SIGN 4

#define OSSL\_OP\_SIGN 5

#define OSSL\_OP\_ASYM\_KEYGEN 6

#define OSSL\_OP\_ASYM\_PARAMGEN 7

#define OSSL\_OP\_ASYM\_ENCRYPT 8

#define OSSL\_OP\_ASYM\_SIGN 9

#define OSSL\_OP\_ASYM\_DERIVE 10

如果提供者(Provider)要使算法可用于libcrypto，它必须注册一个查询回调的操作，该操作在给定操作标识的情况下返回一个实现描述符数组：

< algorithm name, property definition string, implementation `OSSL\_DISPATCH\*` >

因此，如果给定的操作是OSSL\_OP\_DIGEST。

算法由字符串标识。

核心(Core)为提供者提供了一组以函数表的形式使用的服务。

提供者(Provider)还将提供用于返回信息的服务(以附录2-参数传递)通过提供者(Provider)提供的回调(例如：

版本号

构建字符串-根据当前OpenSSL相关的构建信息(仅在提供者(Provider)级别)

提供者(Provider)名称

单个操作可能需要定义多个函数回调才能实现该操作。每个函数将由一个数字函数标识标识。每个标识对于操作和函数的组合都是唯一的，即分配给摘要操作的init函数的编号不能再用于其他操作的init函数，它们将有自己的唯一编号。例如，摘要操作需要以下这些功能：

#define OSSL\_OP\_DIGEST\_NEWCTX\_FUNC 1

#define OSSL\_OP\_DIGEST\_INIT\_FUNC 2

#define OSSL\_OP\_DIGEST\_UPDATE\_FUNC 3

#define OSSL\_OP\_DIGEST\_FINAL\_FUNC 4

#define OSSL\_OP\_DIGEST\_FREECTX\_FUNC 5

typedef void \*(\*OSSL\_OP\_digest\_newctx\_fn)(void \*provctx);

typedef int (\*OSSL\_OP\_digest\_init\_fn)(void \*ctx);

typedef int (\*OSSL\_OP\_digest\_update\_fn)(void \*ctx, void \*data, size\_t len);

typedef int (\*OSSL\_OP\_digest\_final\_fn)(void \*ctx, void \*md, size\_t mdsize,

size\_t \*outlen);

typedef void (\*OSSL\_OP\_digest\_freectx\_fn)(void \*ctx);

对于无法处理多部分操作的设备，也建议使用“全在一个”版本：

#define OSSL\_OP\_DIGEST\_FUNC 6

typedef int (\*OSSL\_OP\_digest)(void \*provctx,

const void \*data, size\_t len,

unsigned char \*md, size\_t mdsize,

size\_t \*outlen)

然后，提供者(Provider)定义数组，其中包含每个算法实现的一组函数和每个操作的一个算法描述符数组。上面提到了算法描述符，并将公开定义如下：

typedef struct ossl\_algorithm\_st {

const char \*name;

const char \*properties;

OSSL\_DISPATCH \*impl;

} OSSL\_ALGORITHM;

例如(这只是一个例子，提供者(Provider)可以以任何他们想要的方式排列这些东西，重要的是算法查询函数，例如fips\_query\_operationFIPS模块可以为SHA 1算法定义如下数组：

static OSSL\_DISPATCH fips\_sha1\_callbacks[] = {

{ OSSL\_OP\_DIGEST\_NEWCTX\_FUNC, fips\_sha1\_newctx },

{ OSSL\_OP\_DIGEST\_INIT\_FUNC, fips\_sha1\_init },

{ OSSL\_OP\_DIGEST\_UPDATE\_FUNC, fips\_sha1\_update },

{ OSSL\_OP\_DIGEST\_FINAL\_FUNC, fips\_sha1\_final },

{ OSSL\_OP\_DIGEST\_FUNC, fips\_sha1\_digest },

{ OSSL\_OP\_DIGEST\_FREECTX\_FUNC, fips\_sha1\_freectx },

{ 0, NULL }

};

static const char prop\_fips[] = "fips";

static const OSSL\_ALGORITHM fips\_digests[] = {

{ "sha1", prop\_fips, fips\_sha1\_callbacks },

{ "SHA-1", prop\_fips, fips\_sha1\_callbacks }, /\* alias for "sha1" \*/

{ NULL, NULL, NULL }

};

FIPS提供者(Provider) init模块入口点函数可能如下所示：

static int fips\_query\_operation(void \*provctx, int op\_id,

const OSSL\_ALGORITHM \*\*map)

{

\*map = NULL;

switch (op\_id) {

case OSSL\_OP\_DIGEST:

\*map = fips\_digests;

break;

}

return \*map != NULL;

}

#define param\_set\_string(o,s) do { \

(o)->buffer = (s); \

(o)->data\_type = OSSL\_PARAM\_UTF8\_STRING\_PTR; \

if ((o)->result\_size != NULL) \*(o)->result\_size = sizeof(s); \

} while(0)

static int fips\_get\_params(void \*provctx, OSSL\_PARAM \*outparams)

{

while (outparams->key != NULL) {

if (strcmp(outparams->key, "provider.name") == 0) {

param\_set\_string(outparams, "OPENSSL\_FIPS");

} else if if (strcmp(outparams->key, "provider.build") == 0) {

param\_set\_string(outparams, OSSL\_FIPS\_PROV\_BUILD\_STRING);

}

}

return 1;

}

OSSL\_DISPATCH provider\_dispatch[] = {

{ OSSL\_FUNC\_PROVIDER\_QUERY\_OPERATION, fips\_query\_operation },

{ OSSL\_FUNC\_PROVIDER\_GET\_PARAMS, fips\_get\_params },

{ OSSL\_FUNC\_PROVIDER\_STATUS, fips\_get\_status },

{ OSSL\_FUNC\_PROVIDER\_TEARDOWN, fips\_teardown },

{ 0, NULL }

};

static core\_put\_error\_fn \*core\_put\_error = NULL;

static core\_get\_params\_fn \*core\_get\_params = NULL;

int OSSL\_provider\_init(const OSSL\_PROVIDER \*provider,

const OSSL\_DISPATCH \*in,

const OSSL\_DISPATCH \*\*out

void \*\*provider\_ctx)

{

int ret = 0;

/\*

\* Start with collecting the functions provided by the core

\* (we could write it more elegantly, but ...)

\*/

while (in->func\_num > 0) {

switch (in->func\_num) {

case OSSL\_FUNC\_ERR\_PUT\_ERROR:

core\_put\_error = in->func;

break;

case OSSL\_FUNC\_GET\_PARAMS:

core\_get\_params = in->func;

Break;

}

in++;

}

/\* Get all parameters required for self tests

\* 注释：获取自我测试所需的所有参数

\*/

{

/\*

\* All these parameters come from a configuration saying this:

\* 注释：所有这些参数都来自于这样一个配置:

\*

\* [provider]

\* selftest\_i = 4

\* selftest\_path = "foo"

\* selftest\_bool = true

\* selftest\_name = "bar"

\*/

OSSL\_PARAM selftest\_params[] = {

{ "provider.selftest\_i", OSSL\_PARAM\_NUMBER,

&selftest\_i, sizeof(selftest\_i), NULL },

{ "provider.selftest\_path", OSSL\_PARAM\_STRING,

&selftest\_path, sizeof(selftest\_path), &selftest\_path\_ln },

{ "provider.selftest\_bool", OSSL\_PARAM\_BOOLEAN,

&selftest\_bool, sizeof(selftest\_bool), NULL },

{ "provider.selftest\_name", OSSL\_PARAM\_STRING,

&selftest\_name, sizeof(selftest\_name), &selftest\_name\_ln },

{ NULL, 0, NULL, 0, NULL }

}

core\_get\_params(provider, selftest\_params);

}

/\* Perform the FIPS self test - only return params if it succeeds.

\* 注释：执行FIPS自检——如果成功，只返回参数。

\*/

if (OSSL\_FIPS\_self\_test()) {

\*out = provider\_dispatch;

return 1;

}

return 0;

}

### 算法选择

多个提供者(Provider)在任何时候都可能可用。为该版本重新编译的现有应用程序代码应该继续工作。同时，只要稍微调整代码，就可以使用新的基于属性的算法查找功能来查找和使用算法。

为了说明这是如何工作的，下面的代码是一个示例，说明如何使用OpenSSL 1.1.1完成简单的AES-CBC-128加密。为了简单起见，所有的错误处理都被删除了。

EVP\_CIPHER\_CTX \*ctx;

EVP\_CIPHER \*ciph;

ctx = EVP\_CIPHER\_CTX\_new();

ciph = EVP\_aes\_128\_cbc();

EVP\_EncryptInit\_ex(ctx, ciph, NULL, key, iv);

EVP\_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);

EVP\_EncryptFinal\_ex(ctx, ciphertext + clen, &clentmp);

clen += clentmp;

EVP\_CIPHER\_CTX\_free(ctx);

在OpenSSL3.0中，这样的代码将继续工作，并使用提供者(Provider)的算法(假设没有配置任何其他代码，它将是默认的提供者(Provider))。它还可以使用显式获取重写，如下所示。显式获取还允许应用程序在需要时指定非默认库上下文(osslctx(在本例中)：

EVP\_CIPHER\_CTX \*ctx;

EVP\_CIPHER \*ciph;

ctx = EVP\_CIPHER\_CTX\_new();

ciph = EVP\_CIPHER\_fetch(osslctx, "aes-128-cbc", NULL); /\* <=== \*/

EVP\_EncryptInit\_ex(ctx, ciph, NULL, key, iv);

EVP\_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);

EVP\_EncryptFinal\_ex(ctx, ciphertext + clen, &clentmp);

clen += clentmp;

EVP\_CIPHER\_CTX\_free(ctx);

EVP\_CIPHER\_free(ciph); /\* <=== \*/

应用程序可能希望使用来自不同提供者(Provider)的算法。

例如，假设应用程序希望使用FIPS提供者(Provider)的一些算法，但在某些情况下仍然使用默认算法。这可以通过不同的方式来实现。

只使用FIPS算法。

默认使用FIPS算法。能够在“根据需要”的基础上重写它，以获得对非FIPS算法的访问权。

默认情况下不关心FIPS算法。能够在“视需要”的基础上重写它以获得FIPS算法。

### 只有FIPS

与为预3.0.0 OpenSSL编写的代码相比，只需要实现FIPS所需做的事情如下：

int main(void)

{

EVP\_set\_default\_alg\_properties(NULL, "fips=yes"); /\* <=== \*/

...

}

然后，上面的加密代码使用EVP\_aes\_128\_cbc()会像以前一样继续工作。这个EVP\_EncryptInit\_ex()调用将使用这些默认的算法属性，然后通过核心(Core)查找它，以便获得FIPS实现的句柄。然后，该实现将与EVP\_CIPHER\_CTX对象。如果没有合适的算法实现可用，则EVP\_Encrypt\_init\_ex()打电话就会失败。

EVP\_set\_default\_alg\_properties的第一个参数是库上下文，NULL是默认的内部上下文。

### 默认为FIPS，但允许重写

默认使用FIPS算法，但在视需要而定基于非FIPS算法，与为预3.0.0 OpenSSL编写的代码相比，应用程序可能会这样做：

int main(void)

{

EVP\_set\_default\_alg\_properties(osslctx, "fips=yes"); /\* <=== \*/

...

}

EVP\_CIPHER\_CTX \*ctx;

EVP\_CIPHER \*ciph;

ctx = EVP\_CIPHER\_CTX\_new();

ciph = EVP\_CIPHER\_fetch(osslctx, "aes-128-cbc", "fips!=yes"); /\* <=== \*/

EVP\_EncryptInit\_ex(ctx, ciph, NULL, key, iv);

EVP\_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);

EVP\_EncryptFinal\_ex(ctx, ciphertext + clen, &clentmp);

clen += clentmp;

EVP\_CIPHER\_CTX\_free(ctx);

EVP\_CIPHER\_free(ciph); /\* <=== \*/

在这里EVP\_CIPHER\_fetch()调用将合并以下属性：

默认算法属性

作为参数传入的属性(其中传递的属性优先)。

因为EVP\_CIPHER\_fetch()调用覆盖默认的“FIPS”属性，它将查找AES-CBC-128的实现，而不是“FIPS”。

在这个例子中，我们看到一个非默认的库上下文正在被使用.这只有在显式提取的实现中才有可能。

(注意事项：“fips!=yes"不如"fips=no”，但在这里提供了“不等于”运算符的示例)。

默认情况下不关心FIPS并允许覆盖FIPS

默认为不使用FIPS算法，但在视需要而定在使用FIPS的基础上，与为预3.0.0 OpenSSL编写的代码相比，应用程序代码可能如下所示：

EVP\_CIPHER\_CTX \*ctx;

EVP\_CIPHER \*ciph;

ctx = EVP\_CIPHER\_CTX\_new();

ciph = EVP\_CIPHER\_fetch(osslctx, "aes-128-cbc", "fips=yes"); /\* <=== \*/

EVP\_EncryptInit\_ex(ctx, ciph, NULL, key, iv);

EVP\_EncryptUpdate(ctx, ciphertext, &clen, plaintext, plen);

EVP\_EncryptFinal\_ex(ctx, ciphertext + clen, &clentmp);

clen += clentmp;

EVP\_CIPHER\_CTX\_free(ctx);

EVP\_CIPHER\_free(ciph); /\* <=== \*/

在这个版本中，我们没有覆盖“main”中的默认算法属性，因此您将得到标准的开箱即用的默认值，即不强制使用FIPS。但是，我们已经显式地将“FIPS”属性设置在EVP\_CIPHER\_fetch()级别，以便重写默认设置。什么时候EVP\_CIPHER\_fetch()使用核心(Core)查找算法，它将获得FIPS的引用(如果没有这样的算法，则失败)。

非对称算法选择

注意，对于对称加密/解密和消息摘要，可以使用现有的OpenSSL对象来表示算法，即EVP\_CIPHER和EVP\_MD。对于非对称算法，不存在等效对象。使用的算法是隐式地从EVP\_PKEY的类型推断出来的。为了解决这一问题，将引入一种新的非对称算法对象。在下面的示例中，执行ECDH密钥派生。我们使用一个新的算法对象EVP\_ASYM查找FIPS ECDH实现(当然，假设我们知道给定的私钥是一个ECC密钥):

EVP\_PKEY\_CTX \*pctx = EVP\_PKEY\_CTX\_new(privkey, NULL);

EVP\_ASYM \*asym = EVP\_ASYM\_fetch(osslctx, EVP\_PKEY\_EC, "fips=yes");

EVP\_PKEY\_CTX\_set\_alg(pctx, asym));

EVP\_PKEY\_derive\_init(pctx);

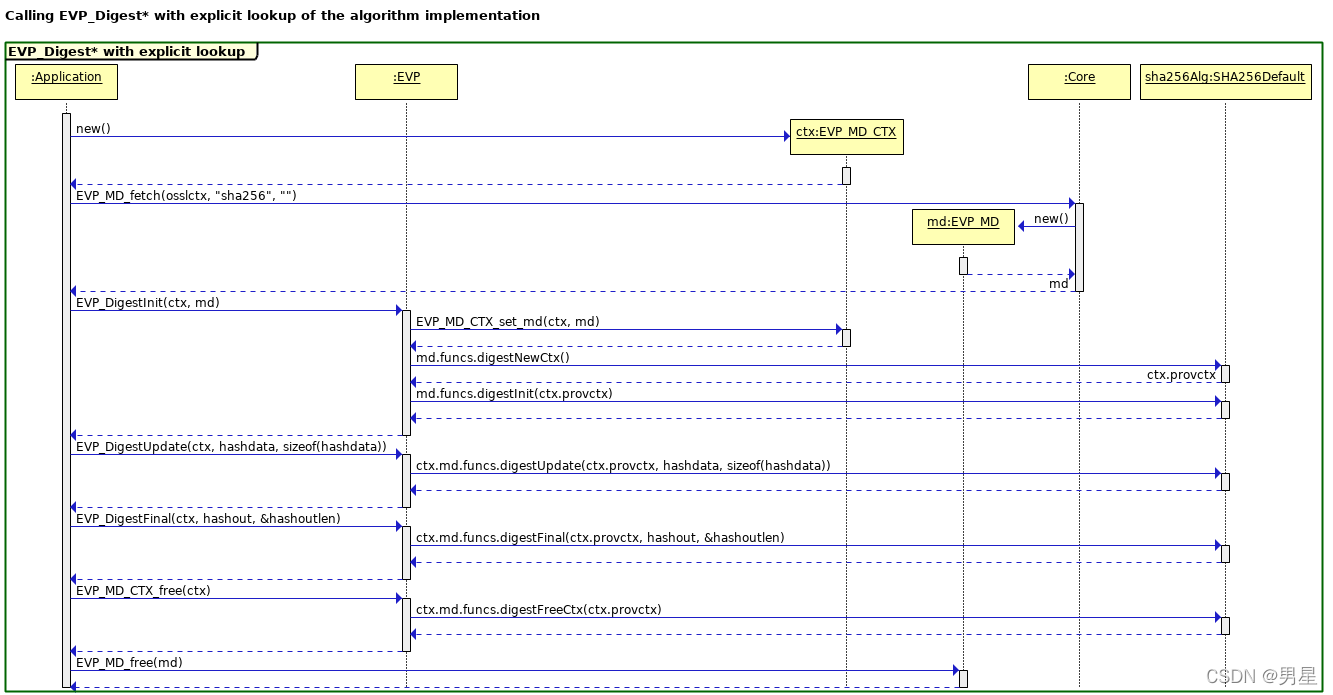
EVP\_PKEY\_derive\_set\_peer(pctx, pubkey);

EVP\_PKEY\_derive(pctx, out, &outlen);

EVP\_PKEY\_CTX\_free(pctx);

算法选择的示例动态视图

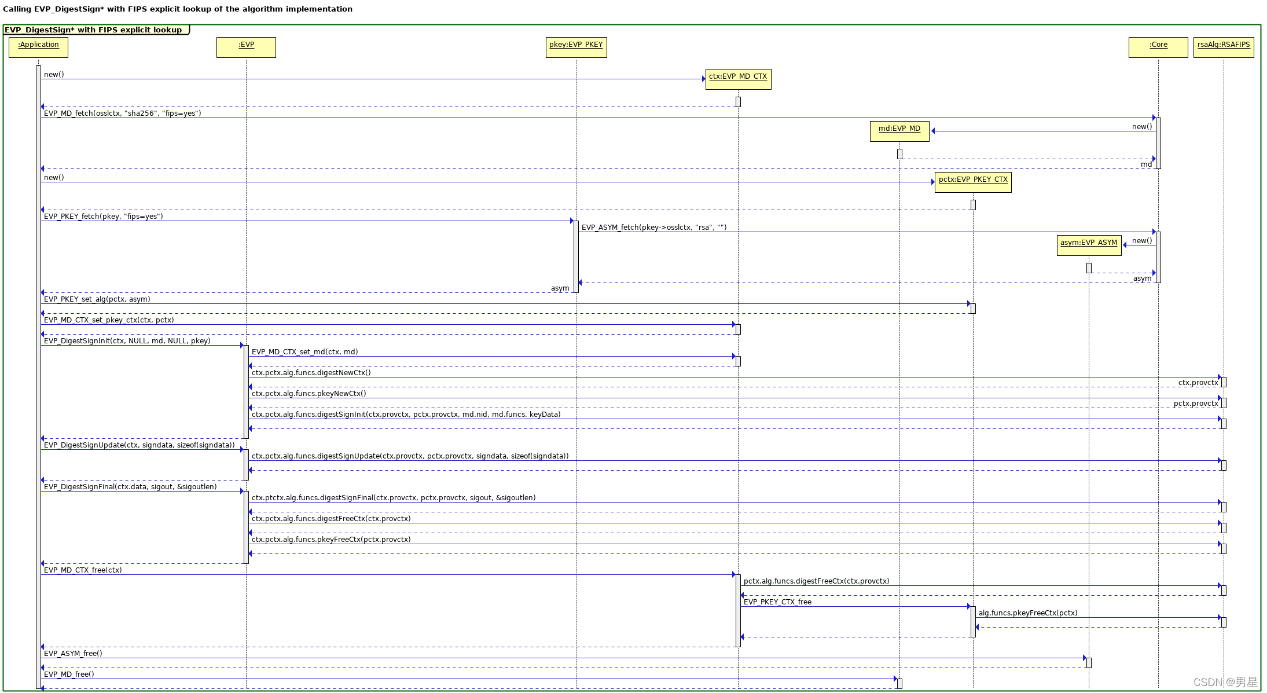
下面的序列图显示了如何从默认提供者(Provider)中选择和调用SHA 256算法的示例。



请注意，每个EVP层调用都是由EVP层中的外层封装实现的，它根据算法在提供者(Provider)中调用类似命名的函数。要使用的特定提供者(Provider)函数将通过显式的方式在核心调度表(Core Dispatcher)表中查找EVP\_MD\_fetch()调用，该调用将消息摘要名称指定为字符串和任何其他相关属性。返回的“md”对象包含指向所选提供者(Provider)中算法实现的函数指针。

这个EVP\_MD\_CTX对象不会传递给提供者，因为我们不知道是否有任何特定的提供者(Provider)模块被链接到libcrypto。相反，我们只是通过一个黑匣子句柄(void \*(指针)，该提供者(Provider)将与它所需的任何结构相关联。这是在显式分配期间分配的。digestNewCtx()在操作开始时调用提供者(Provider)，并在结束时调用digestFreeCtx()释放。

下一个图表显示了一个稍微复杂的场景，即EVP\_DigestSign操作使用RSA和SHA 256。此图是从libcrypto的角度绘制的，算法由FIPS模块提供。后面的一节将从FIPS模块的角度来研究这个场景。



EVP\_DigestSign操作更复杂，因为它涉及两种算法：签名算法和摘要算法。通常，这两种算法可能来自不同的提供者或相同的提供者。对于FIPS模块，算法必须来自同一个FIPS模块提供者(Provider)。如果尝试进行其他操作，则操作将失败。

尽管有两个算法增加了复杂性，但同样的概念适用于更简单的算法。EVP\_Digest操作，如前面的图表所示。产生了两个上下文：EVP\_MD\_CTX和一个EVP\_PKEY\_CTX。这两种方法都不会传递给提供者(Provider)。相反，黑匣子(void \*)句柄通过显式“newCtx”提供者(Provider)调用创建，然后在随后的“init”, “update” 和 “final” 操作中传递这些句柄。

预先使用显式的EVP\_MD\_fetch()和EVP\_ASYM\_fetch()调用在核心调度表中查找算法。

# Provider简介

[《OpenSSL3.0学习之四 提供程序简介｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122701885?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十 提供程序-base｜CSDN创作打卡》\_男星的博客-CSDN博客\_openssl 死锁](https://blog.csdn.net/include_he/article/details/122705688?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十一 提供程序-digest｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122785376?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十二 提供程序-cipher｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122785395?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十三 提供程序-rand｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122785674?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十四 提供程序-keymgmt｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122792427?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十五 提供程序-kem｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122792663?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十六 提供程序-asym\_cipher｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122792748?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十八 提供程序-signature｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122835900?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之十九 提供程序-mac｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122836731?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十 提供程序-kdf》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122837403?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十一 提供程序-encoder》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122838384?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十二 提供程序-decoder》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122839824?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十三 提供程序-storemgmt》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122840331?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十四 提供程序-object》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122840692?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十五 EVP简介》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122702278?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十六 EVP\_SealXXX》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/123253151?spm=1001.2014.3001.5502)

[《OpenSSL3.0学习之二十七 EVP\_OpenXXX》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/123259282?spm=1001.2014.3001.5502)

[《OpenSSL3.0编程之二 官方示例-digest》\_男星的博客-CSDN博客\_openssl 示例](https://blog.csdn.net/include_he/article/details/123406240?spm=1001.2014.3001.5502)

# 基本Provider

[《OpenSSL3.0学习之五 OpenSSL 基本提供程序｜CSDN创作打卡》\_男星的博客-CSDN博客](https://blog.csdn.net/include_he/article/details/122701935?spm=1001.2014.3001.5502)

# 默认Provider

[《OpenSSL3.0学习之六 OpenSSL 默认提供程序｜CSDN创作打卡》\_男星的博客-CSDN博客\_openssl 计算cmac](https://blog.csdn.net/include_he/article/details/122702006?spm=1001.2014.3001.5502)