**1.非功能性需求NFR指标的定义**

有趣的是，软件架构基本上与功能性需求无关。我们可以用任何软件架构实现一组用例（应用程序的功能性需求）。但是软件架构会对非功能性需求有直接的影响。

因此我们在做软件的架构选型的时候，要考虑能否满足系统的非功能性需求。

主要的非功能性需求有：



我们来看下这些非功能性需求指标的定义。

* **NFR-1健壮性** 软件可以对异常条件进行适当的反应的能力。
* **NFR-2可扩展性** 软件产品适应需求变化的容易程度。
* **NFR-3可重用性** 软件组件可以在许多不同应用程序中作为组件的能力。
* **NFR-4兼容性** 软件的不同组件互相组合的容易程度。
* **NFR-5有效性** 对硬件资源如处理器性能、存储空间、带宽等要求越低效率越高。
* **NFR-6可移植性** 将软件产品迁移到各种硬件和软件环境运行的容易程度。
* **NFR-7易用性** 不同背景和资质的人能够学会使用软件产品，并用来解决实际问题的容易程度。它还包括安装、运维和监控的容易程度。
* **NFR-8及时性** 当用户需要时，软件系统能够及时发布的能力。
* **NFR-9可测试性** 软件发现故障并隔离、定位其故障的能力。

**2.边缘计算软件的非功能需求**

接下来，我们把边缘计算的非功能性需求简称为ECNFR。我们来看下边缘计算场景下，非功能性需求的具体要求：

* **ECNFR-1健壮性** 边缘计算软件所处工业现场环境恶劣，断网、断电经常发生。因此需要有对异常条件进行相应处理的能力。
* **ECNFR-2可扩展性** 边缘计算客户需求多变，现场有各种协议设备，需要连接到各种软件或云。因此需要有很强的适应能力。可以让不同技能的开发团队松耦合的独立开发不同的模块。
* **ECNFR-3可重用性** 边缘计算软件的组件可以在许多不同应用场合重用，这样才能提高实施效率，减少开发工作量。比如工业互联网领域，最常见的需求就是设备协议解析模块和云端连接模块要能够重用。
* **ECNFR-4兼容性** 边缘计算软件的各种定制组件与可重用基础组件组合后，可形成完整的功能来满足不同客户需求。不同功能模块可以组合，降低开发时间。
* **ECNFR-5有效性** 边缘计算现场的运行环境的硬件资源大小不一。边缘计算软件要能根据硬件环境进行伸缩，在不同配置的硬件环境上都能运行。
* **ECNFR-6可移植性** 边缘计算软件要能够容易地移植到不同的软硬件环境上运行。可以独立部署边缘计算软件的其中一个或多个功能模块，从而可以更灵活地部署。比如添加了一个针对特定型号PLC的设备数据采集模块，要可以方便的移植这个服务到现场的嵌入式硬件上，而不需要部署所有其他的模块。同时因为部署的操作系统环境和虚拟化方式不同，不能限制用户使用哪种容器部署方式。此外如果边缘计算软件限制用特定的语言，会减少社区开发人员的用户数量。因此边缘计算软件架构不能限制开发人员用哪种语言，从而加速新技术导入，增加解决方案的可选项。比如数据采集模块由于要实时性高，运行的计算节点资源有限，开发人员就可以用C语言开发。对于机器学习数据分析功能，则可以用Java、Python等开发。
* **ECNFR-7易用性** 使用边缘计算软件的用户背景和资质不同，不光IT开发人员要能用，甚至工控工程师和嵌入式工程师都要会在上面开发。要让他们都能够学会使用，并用来解决实际问题。
* **ECNFR-8及时性** 边缘计算软件面向的行业需求众多，需求差异性很大，要能够让开发团队更快更容易地交付新产品、新功能。边缘计算软件架构不能要求开发人员理解整个系统全貌才能开发。开发人员只需要理解单个组件的接口，就可以开发。从而降低开发人员的门槛，同时可以让不同组件并行开发。
* **ECNFR-9可测试性** 边缘计算软件要能够容易地发现、定位并隔离故障。降低了代码实现时引入Bug的风险。这样可以持续交付质量稳定的功能模块给客户。

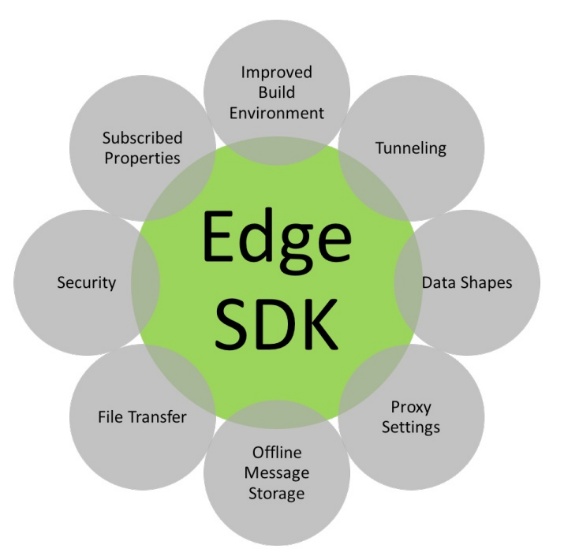
**3.边缘计算软件的架构盘点**

和设计飞机驾驶舱一样，要设计好边缘计算软件，必须要让系统（边缘计算软件）适应个体（客户需求），而不是根据个体的平均值来设计系统。**边缘计算软件要能调整各个组成部分，从而适应不同的客户需求。**具体实现时，有许多种方式：

* 单体程序（一个独立的大程序）。
* 面向对象的C，其中许多业务组件都可以继承扩展。
* 数据采集、数据处理、数据上传插件化，每个插件模块封装成一个动态链接库。
* OSGI模式。比如Kura和Bosch IoT。
* 微服务模式，也就是EdgeX选择的架构方式。
* Lambda@Edge。例如AWS Greengrass。

**单体程序**

PTC的Thingworx Edge SDK就属于这种。采用模块化的方式开发。它的C 语言版本的SDK在一个C项目中提供了与云端的安全连接、数据传输压缩、Edge Extension、文件传输、元数据、VNC和SSH隧道、代理服务器认证、离线消息存储等功能。



它虽然有C、Java、.NET等多种语言的SDK开发包，但是二次开发人员需要在它基础上开发。开发前需要理解整个代码的结构。

这种模式适合商业化公司开发自己家的边缘计算SDK连到自己家的云上。但是不能让不同团队在上面分开开发不同模块。单体系统有它的优点：

* 易于开发。现代的IDE设计初衷就是可以管理单个的复杂大项目。
* 易于进行大的改动。既然所有代码都在一个IDE里，可以在IDE里连数据库改Schema，在IDE里导入新的库，在IDE里改代码，然后在IDE里连到版本控制系统提交代码。
* 易于测试。改完代码部署后，直接运行单元测试和集成测试脚本。
* 易于部署。在IDE里调用脚本自动构建和部署，或者直接在IDE里把部署好的完整的程序包放到应用服务器里运行即可。

但是单体系统也有缺点：

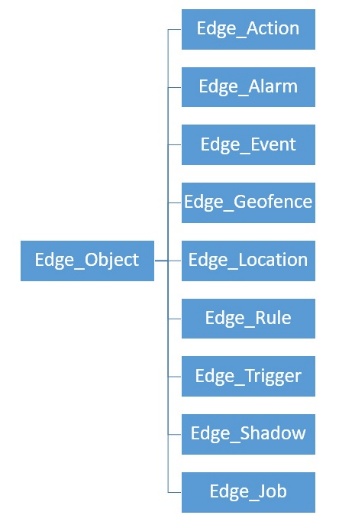
* 过于复杂难于理解。要改动一点代码需要搞懂牵连的许多其他模块的代码。有时候改了几行代码就把别的功能改坏了。许多时候搞不明白的情况下在时间压力下硬着头皮去改，结果代码就越改越乱。代码越积越多, 层次越做越深, 逻辑复杂, 结构混乱, 牵一发而动全身, 说好的高内聚, 松耦合几乎做不到。
* 开发慢。由于复杂性导致改动一点点就需要耗费许多时间。
* 部署测试时间长。测试的时候，哪怕改了一行代码，也要跑全部测试，以免引入bug。构建一个大项目所需的时间也更长。由于程序很大，部署测试版本的时间和程序启动时间也很长。
* 难于扩展提升性能。不同模块之间的服务器性能要求是冲突的。比如内存数据库模块需要部署在内存优化的服务器上，对CPU没有太高要求。而机器学习模块需要的是CPU核数多的服务器，对内存没有太大要求。
* 软件质量难以保障。每次升级版本，哪怕改动了几行代码，为了软件质量，也要进行完整详尽的测试。但是实际上一般软件公司很难做到这一点。对于大而全的软件，这样不做完整详尽的测试，软件质量就很难保证。
* 被过时的软件技术栈锁定。大而全的软件设计之初的时候，技术栈的选型一般会是当时比较先进主流的技术。但是过了几年后，所用的技术栈就会慢慢过时。过了几年要用一些新技术上去，会很困难甚至不可行，因为里面会有很多依赖关系，改动起来会很困难。

这种情况下，只有纪律性强的团队才能维护好这种单体软件系统。他们会把整个系统模块化，然后写大量的自动化测试代码进行高覆盖。但是他们还是没法解决许多人在一个大系统上进行同步并行开发带来的各种问题。对于过时的技术栈也没有办法。

随着时间的推移，大多数开发团队通常很难保持良好的模块化结构，使得更难以保持应该只影响该模块中的一个模块的更改。扩展需要扩展整个应用程序。

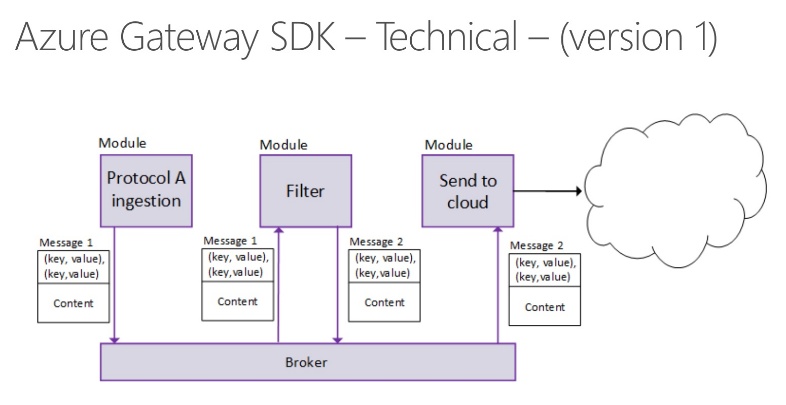
**面向对象C**

这种其实本质上也是一个单体程序。扩展性也不好。对开发人员要求也高。



**插件化**

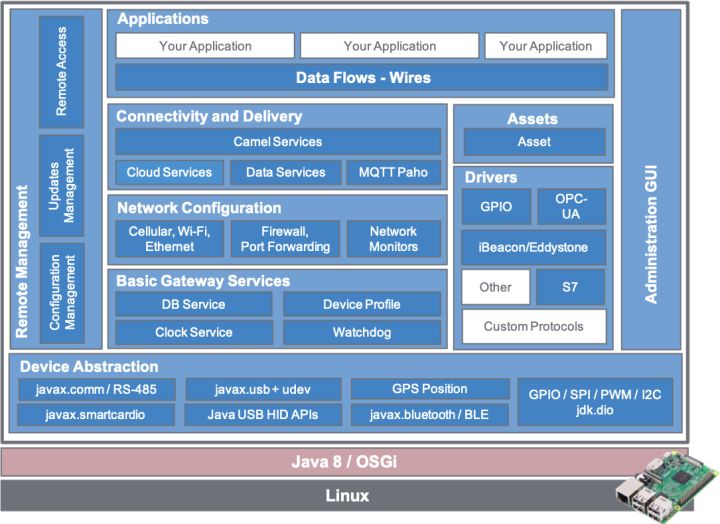
把数据采集、数据分析、数据上传云等容易受到需求变更影响的模块插件化，封装成DLL或so文件。如果有新的需求，开发新的插件。然后修改配置文件，加载新的插件。Azure IoT Edge SDK V1版本就是把协议解析、协议数据处理和数据上云模块化的设计思路。



**OSGI**

Java领域有现成的模块化方案OSGi标准。Eclipse Kura就是基于OSGi标准实现的边缘计算软件框架。

Kura运行在Java虚拟机（JVM）之上，并利用OSGi框架来简化编写可重用软件的过程。Kura API提供对底层硬件的轻松访问，包括串口，GPS，看门狗，USB，GPIO，I2C等。它还提供Web管理界面，以简化本地和远程的配置管理。



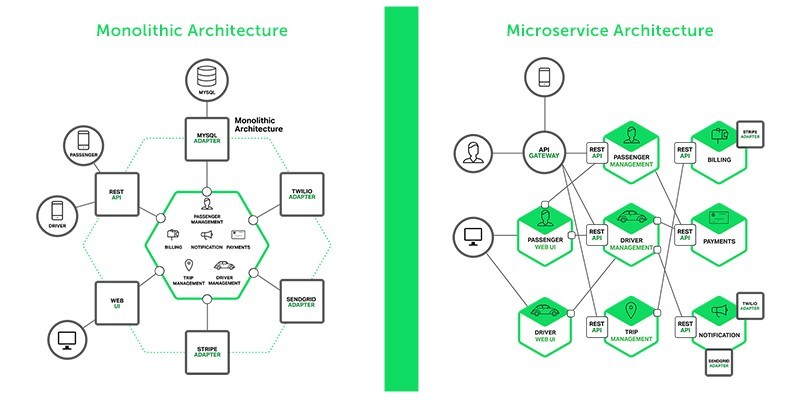
选择OSGi开发有哪些好处呢？

* 降低复杂性 - 使用OSGi技术开发意味着开发Bundle包：OSGi组件。Bundle包不仅减少了错误的数量，而且还使捆绑包更容易开发，因为捆绑包通过定义良好的接口实现了一项功能。
* 重用 - OSGi组件模型使得在应用程序中使用许多第三方组件变得非常容易。越来越多的开源项目为OSGi提供了准备好的JAR。
* 动态管理 - OSGi组件模型是动态模型。可以安装，启动，停止，更新和卸载软件包，而无需关闭整个系统。
* 轻松部署 - OSGi技术不仅仅是组件的标准。它还指定了如何安装和管理组件。许多Bundle包使用此API来提供管理代理。
* 动态更新 -自适应 - OSGi组件模型从头开始设计，允许组件的混合和匹配。这种动态服务模型允许捆绑包找出系统上可用的功能并调整它们可以提供的功能。这使代码更灵活，更灵活。
* 版本控制 - OSGi技术解决了JAR地狱问题。在OSGi环境中，所有Bundle包都经过版本化，只有可以协作的捆绑包在同一个类空间中连接在一起。
* 简单 - OSGi API非常简单。核心API只有一个包，少于30个类/接口。这个核心API足以编写捆绑包，安装它们，启动，停止，更新和卸载它们，并包括所有监听器和安全类。
* 小巧 - OSGi第4版框架可以在大约300KB的JAR文件中实现。对于通过包含OSGi添加到应用程序的功能量来说，这是一个很小的开销。因此，OSGi可以在很多设备上运行：从小型到小型，再到大型机。它只要求运行一个最小的Java VM。

但是使用OSGI框架有个问题，就是限定了开发人员只能用Java开发，而且要熟悉OSGi框架和JVM。

**微服务**

微服务架构风格是一种将单个应用程序开发为一套小型服务的方法，每个小型服务都在自己的进程中运行，并通过轻量级机制（通常是HTTP API）进行通信。这些服务围绕业务功能构建，可通过自动部署机制独立部署。这些服务可以用不同的编程语言编写，并使用不同的数据存储技术。



微服务有如下好处：

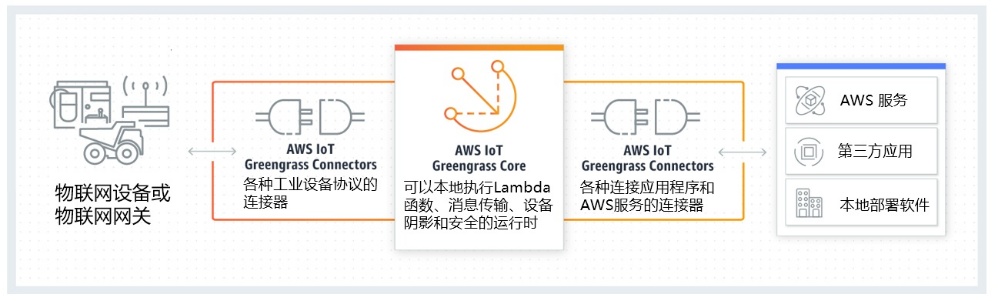
* 敏捷开发 允许组织更快更容易地交付新产品、新功能。
* 可组合性 降低开发时间，通过重用服务提供新功能。
* 可理解性 对单个服务更加容易理解，简化了开发计划
* 独立可部署性 服务可以独立部署，可以更快部署，更灵活
* 团队合作：通过以服务为单位进行小团队开发，降低了沟通成本。
* 多语言支持：可以用合适的工具去完成相应的任务，从而加速新技术导入，增加解决方案的可选项。
* 可靠性：降低了技术债务，技术债务会导致架构老化、不可靠的环境。
* 可伸缩性：根据业务需要伸缩。
* 可测试性：降低了代码实现时引入bug的风险

**边缘侧的无服务架构**

云计算厂商将云端的Serverless无服务架构搬到边缘侧。比较典型的是AWS的GreenGrass。

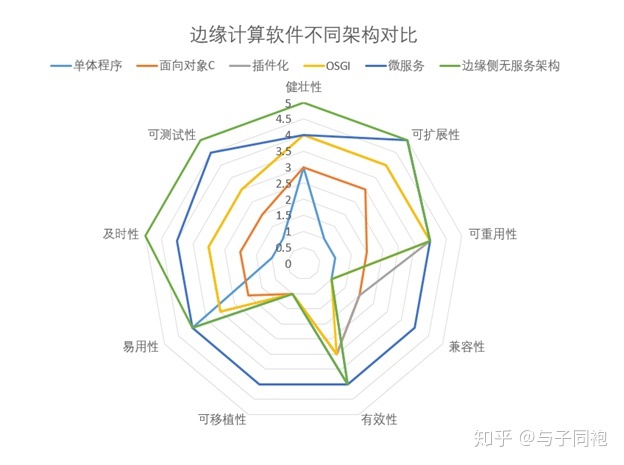
亚马逊AWS的Greengrass相当于离线版本的IoT Core和Lambda函数。

这种无服务架构的问题是运行时较为复杂，可移植性不好。



**4.不同架构方式满足边缘计算非功能性需求的程度**

下图为不同架构方式再边缘计算场景下的非功能性需求的评分雷达图。



从图中可以看出，微服务和边缘侧的无服务架构各方面分数较为均衡。但是无服务架构存在可移植性不好的问题。

因此微服务架构对于边缘计算来说还是很适合的。