Vapula组件开发框架 设计文档

一个完整的应用程序由唯一的主体和众多功能组件组成。

主体负责装载、卸载、调用组件功能，对外提供主要用户界面。

组件对一类组织在一起的功能负责。

组件逻辑上包含多个功能，内部的实现不受到Vapula约束。

但是对外必须提供所有的功能的调用接口。

功能的调用状态、输入输出参数以及其他共享信息（共享给组件和调用方）托管在Vapula提供的私有栈里。

驱动机制

Driver是Vapula的驱动机制的核心所在

驱动开发者可以开发一个具有GetDriverInstance的C接口的DLL

然后实现Library和Invoker基类

Vapula的驱动机制目标是实现所有支持的环境之间两两可以互相调用

Vapula认为所有运行环境加载该环境下的组件是最方便的。

所以驱动开发者需要在驱动DLL中仅开发加载特定运行时的托管过程以及调用Vapula核心Bridge（C++开发的）功能的接口，接口必须是用于特定运行时互操作的（例如PInvoke）

然后驱动开发者按照非C++运行时规范，开发一套调用封装，所有Vapula功能通过互操作调用，然后Vapula核心会将适用于该运行时的组件的加载转交回给运行时去具体实现，这就是Vapula驱动的短路机制。

组件功能调用设计

组件功能通过Invoker、Worker配合，实现调用

Invoker由Library的对应Method的信息通过Driver构造

Invoker的Invoke方法是一整套适应Vapula设计的组件调用过程，包括组件调用前的栈准备、组件调用、异常保护、回滚支持以及调用后的栈维护操作。

用户通过Invoker的Invoke方法可以实现对功能的调用。

强烈建议使用Start、Pause、Resume、Restart、Stop等控制方法去调用并控制。

因为Vapula内部已经实现了一个优化的多核线程池，会按照机器硬件水平去维护一个线程池，然后调度所有的Invoker队列。由于该线程池控制了线程的数量，所以有效降低了应用程序的线程创建、销毁和调度时切换内核态的开销。

线程规模可以设置。

启动时，Worker启动等于逻辑核心数的线程，移入前方（Front）。

Start方法：

将Invoker推送到任务队列，设置Context->State为QUEUE。

当Worker检测到线程空置时，锁定队列，取队首Invoker执行，解锁队列。

Pause方法：

设置控制码，等待Invoker自身主动暂停，如果超时将线程挂起。

将当前线程悬空（移出当前阵营），从后方（Rear）获取一个可用线程，失败则新开一个可用线程，移入前方，并将当前线程移入后方（Rear）。（换血机制）

Resume方法：

检查线程是否挂起，如果挂起则恢复。否则直接切换控制码。

Stop方法：

设置控制码，等待Invoker自身结束。

如果超时，则执行换血机制。

否则在完成后设置线程空置。

Restart方法：

Stop当前的线程。将当前Invoker加入队列。

One worker for one process.

警告：不要把同一个Invoker多次注入工作器，这样会导致一个栈被多个线程竞争，导致不可预测的错误。工作器会检测希望注入的Invoker是否与运行中的某一个相同，这会严重影响效率，可以手工关闭该安全检查。

Runtime设计

Runtime是Vapula的主管理器对象，用于在运行期间维护这些对象：

1 驱动集合

2 切面集合

3 库集合

4 栈集合

5 编织机

6 工作者

加载vf\_bridge库时，

Vapula发展目标

1. 支持虚拟线程以及线程池
2. 支持自定义类型扩展
3. 支持数据全端传输
4. 支持插件化组件模板（？）
5. 支持自动化测试
6. 支持智能（自动化）软件建模
7. 支持全端消息队列和事件
8. 云

**Vapula调用功能的过程**

调用方装载Vapula组件 => 调用方构造Vapula功能的执行器（此步骤会构造私有栈） => 调用方提供输入参数（根据需要） => 调用方通过执行器调用功能 => 调用方获取上下文并监听状态 => 调用方析构执行器（此步骤会销毁私有栈）

**Vapula组件逻辑组成、物理组成、建模系统的关系**

一个Vapula组件是一个动态链接库主体与其自身依赖

一个Vapula组件对应组件建模的图组

一个Vapula功能对应组件建模的图

一个Vapula组件包含若干功能

**Vapula类型系统与用户数据托管方案**

Vapula的类型系统没有与组件关联，所以没有面向对象组件设计的必要，

组件和功能也不能对应到面向对象的类和方法。

不设计基于组件的类型系统，组件与数据类型不存在直接关系

Vapula的类型系统仅用于约束：在多个节点间传递的数据与数据结构信息。

此处节点指组件功能（不确定运行时下的不确定组件的不确定功能）、调用方、非Vapula相关数据源或数据受体

为了实现这样的设计目标，Vapula将所有数据类型分为3类：

1. 裸数据（Raw）

组成：长度，数据

1. 有序值数据（Value）

组成：交错，长度，数据

1. 文本（Text）

组成：长度，（有效内容长度）

目前没有为 数据结构 设计统一的 描述用途 数据类型。

为了在多种运行时环境之间共享这些数据，Vapula约束了数据传递的原则。

1. 所有数据在传输时，同类数据直接复制。
2. 序列化时机
3. 序列化机制（值=>Base64字符串，文本=>字符串，裸数据根据是否需要结构化信息：是=>JSON字符串；否=>Base64字符串）
4. 反序列化使用逆过程

Vapula使用可扩展类型接口提供用户自定义类型的支持。

用户有这样的需求：

1. 用户使用特定运行时环境，该运行时环境自身具备类型系统的约束。用户在该类型系统约束下自定义类型。
2. 用户在某运行时环境下定义的类型在其他运行时环境可用。

为了设计这样的通用类型系统约束，Vapula通过特定的序列化接口进行中间转换。

所有的用户类型在进行传递时

Vapula通过数据集（Dataset）和记录（Record）实现用户数据托管。

数据集是记录的有序集合，提供记录的检索和动态增减功能。

数据集的生成有两种途径，一种是在组件操作中生成，一种是用户自行生成。

当组件加载时，会为每个方法构造原型数据集。

当用户调用组件方法时，调用器会复制原型数据集，并托管在虚拟栈。

用户也可以自行编写符合规范的XML，构造用户自定义信封。

Vapula通过堆（和内存池）管理数据集中托管的记录的数据。

记录是参数的数据实体，由数据集托管。

每个记录包含数据类型、访问模式、数据尺寸等信息。

Vapula支持3种数据类型。

Vapula的访问模式有3种：仅输入、仅输出、输入或输出。

（表格）

通过这些基本类型，Vapula理论上可以操作任意数据类型。

受制于不同语言、运行时对数据类型的定义的差别。

Vapula做出以下限制：

除Object和String外，其他类型为基值类型

String是Object的一种特殊细分，作为非基值类型与object没有本质区别，

只是可以使用一些更具体的API进行操作。

基值类型变量的数据作为一个数组存储

非基值类型变量的数据只能存储为一个独立个体，不能存储为数组

Vapula会识别写入和读取的类型是否是基值类型

根据判定结果，Vapula确定是否可以调用访问方法

同时，Vapula不关心数据的具体存储类型

对于基值类型，Vapula提供一些语法糖（模板函数），

可进行快速单值读取和写入，目的仅仅是简化操作

再次重申，Vapula不检测访问基值类型时使用的数据类型是否匹配

写入字符串时会自动复制并存储副本

投递操作：

投递是Vapula定义的一种行为，表示参数从一个变量传递到另一个变量，可以通过Deliver方法调用。

在Variable之间进行Deliver涉及潜在的内存复制，

不复制内存仅会传递数据的访问权，通过复制操作可以复制数据的拥有权。

Vapula不关心类型的实际数据构成，所以类型之间虽然执行了数据传递，

但是不能保证这个操作是有实际意义的。

为了提供更多灵活性，

Variable还提供地址操作、序列化操作

地址操作包括内存取地址和地址还原。

序列化操作提供内存压缩编码支持。

通过内存压缩编码可以将任意数据块压缩编码成base64，方便数据传输

地址操作通过Read、Write即可实现

一般获得Envelope后：

设置值：

Variable\* var = env -> GetVariable(1);

var -> Write<int>(12);

var -> Write (data, 1024, true);

var -> Write(str, true); //非基值类型可用