美宜佳
MEIYIJIA

开发手册

0.0.1（非正式预览版）

美宜佳 MPOS 系统开放平台

修订：廖　宁
校阅：哈　哈

版权所有 © 东莞彩星信息科技有限公司 2018。保留一切权利。

中国 - 东莞

目录

1 引言

1.1 背景及定义

现 MPOS 系统作为一个独立系统的堆砌，存在一些诸如在扩展性、可靠性和维护成本上的一些问题。为此公司适时决策在开发工具不变的情况下对 MPOS 系统进行一次重构，故推出美宜佳 MPOS 系统开放平台（以下简称“平台”）作为项目的架构（不是框架）。

本手册作为平台的组成部分。对平台的结构和开发流程作了说明，提出了基本的规范和约束条件。

1.2 平台概述

平台是一个基于在提供基础的运行生态(主要包含日志、接口寄存、参数、主题和一种抽象图形接口等等)后，通过加载基本运行库和各业务模块的形式实现系统的运行。各业务接口间可以实现快速、模块化的开发，达到解耦、可插拔、可伸缩的目的。做到信息系统设计的系统性、灵活性、可靠性、经济性和安全性要求。也为将来第三方的接入，以及顺应移动端发展的趋势做好准备。

1.3 开发环境

基于 Free Pascal 和 Lazarus 的 Pascal 语言可视化编程 IDE，版本 6.4；
开发系统 Windows7 旗舰版 x64 sp1；
测试系统 Ubuntu 16.01 LTS x64。

1.4 平台思路

现单体架构随着系统规模的扩大，它暴露出来的问题也越来越多，主要有以下几点：复杂性逐渐变高（所有的开发在一个项目改代码，递交代码相互等待，代码冲突不断）；代码维护难，技术债务逐渐上升；部署不灵活，部署速度逐渐变慢（构建时间长，任何小修改必须重新构建所有牵涉业务，这个过程往往很长。同时也因此造成稳定性下降）；扩展性不够，无法按需伸缩，缺少可控性。

要解决上述问题就要降低程序的耦合性。耦合就是联系，耦合越强，联系越紧密。在程序中紧密的联系并不是一件好的事情，因为两种事物之间联系越紧密，你更换其中之一的难度就越大，扩展功能和 debug 的难度也就越大。所以要减少协同。而程序的各种功能是由许许多多的不同对象协作完成的，所以要模块化。这样，各个模块内部是如何通信对系统而言就不那么重要了。系统只关注模块化间的交互。

所以平台采用类微服务的方式进行面向接口模块化架构，模块间进行轻量级通信。

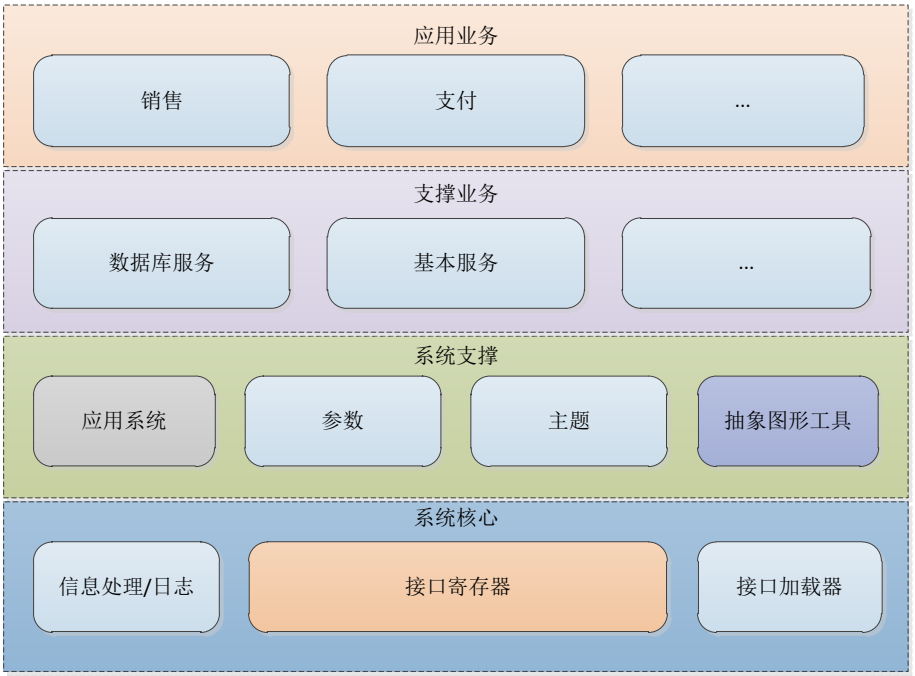
1.5 专有名词

库：指一个 Library 工程。
模块：完成某一特定功能的程序，通常对应一个库，可以包含多个业务。
包：指 IDE 的开发包。

2 体系结构

2.1 平台层次

平台的按依赖和功能性划分为四个层次，如下图所示。其中支撑业务层和应用业务层都属于可拓展变更的业务层次；系统核心层和系统支撑层构成系统的基本生态环境。



2.1.1 应用业务层

作为一个开放式平台可以自由的加入各应用模块，诸如销售、支付等应用业务模块。这些应用业务模块都是用指定的准入规范编写的，并且这些应用业务模块都是可以被开发人员开发的其他应用业务模块所替换，能够做到灵活和个性化。

2.1.2 支撑业务层

应用支撑层是我们进行应用开发的基础，很多核心应用模块也是通过这一层来实现其核心功能的，该层可以简化一些组件的重用，开发人员可以直接使用其提供的组件来进行快速的应用程序开发，也可以通过拓展而实现个性化。

当然不是所有的应用业务层模块都会依赖于支撑业务层的功能，当其实现并不需要所有的业务服务时。

2.1.3 系统支撑层

为整个业务层提供支撑，包括主题、参数等配置管理，抽象图形等工具，提供应用系统的公共功能。

2.1.4 系统核心层

提供接口寄存器、接口加载器对接口进行管理。提供信息处理（日志驱动）。是整个系统运行模块间的重要纽带。

2. 2 工作形式

平台参照 DDD 领域驱动设计采用面向接口的业务模块化工作形式。

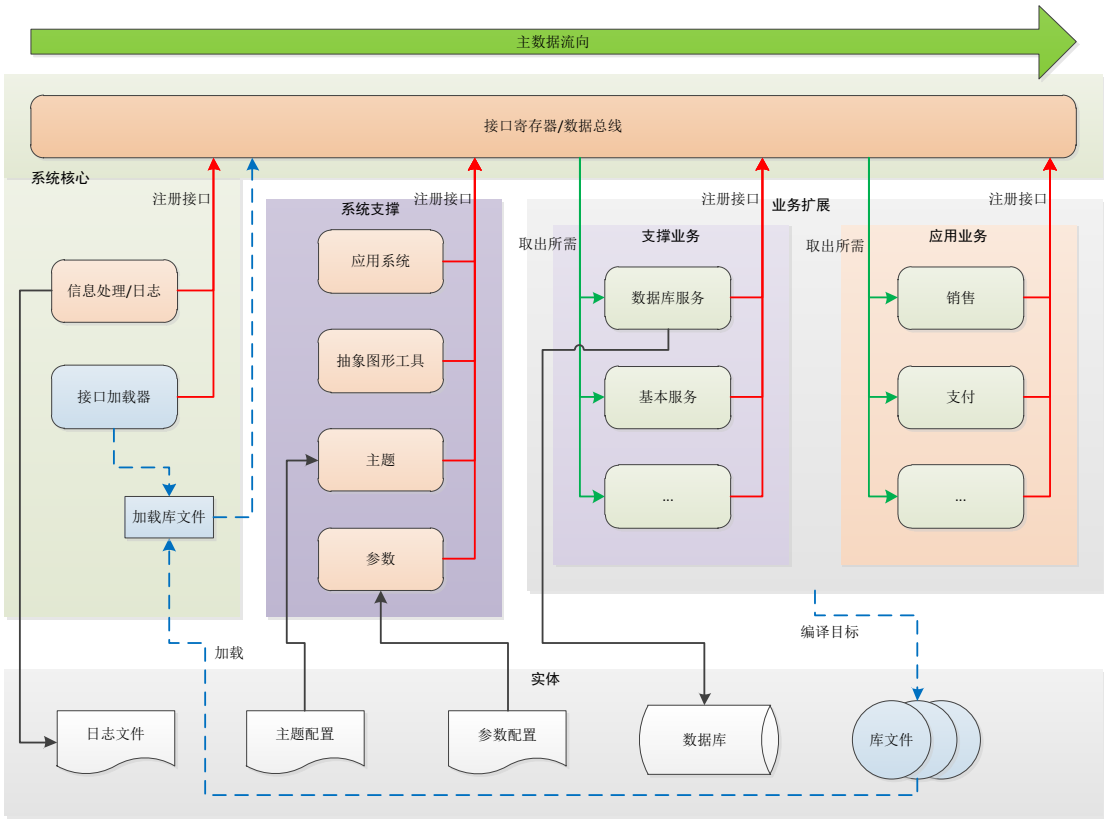
2.2.1 业务模块化

软件架构是一个包含各种组织的系统组织，它们彼此或和环境存在关系。系统架构的目标是解决利益相关者的关注点。诚如康威定律所表：设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。组织沟通方式会通过系统设计表达出来，沟通成本随着团队成员的增加而以几何级数增加。线型系统和线型组织架构间有潜在的异质同态特性。所以要创建独立的子系统（即业务模块化），减少沟通成本。

单个小型的业务功能模块，每个模块进行自己的处理和轻量接口通讯机制，可以部署在单个或多个应用上。业务模块化是一种松耦合的、有一定的有界上下文的面向服务架构。也就是说，如果一个模块进行修改，有模块需要同时修改，那么它们就不是模块化的，因为它们紧耦合在一起。

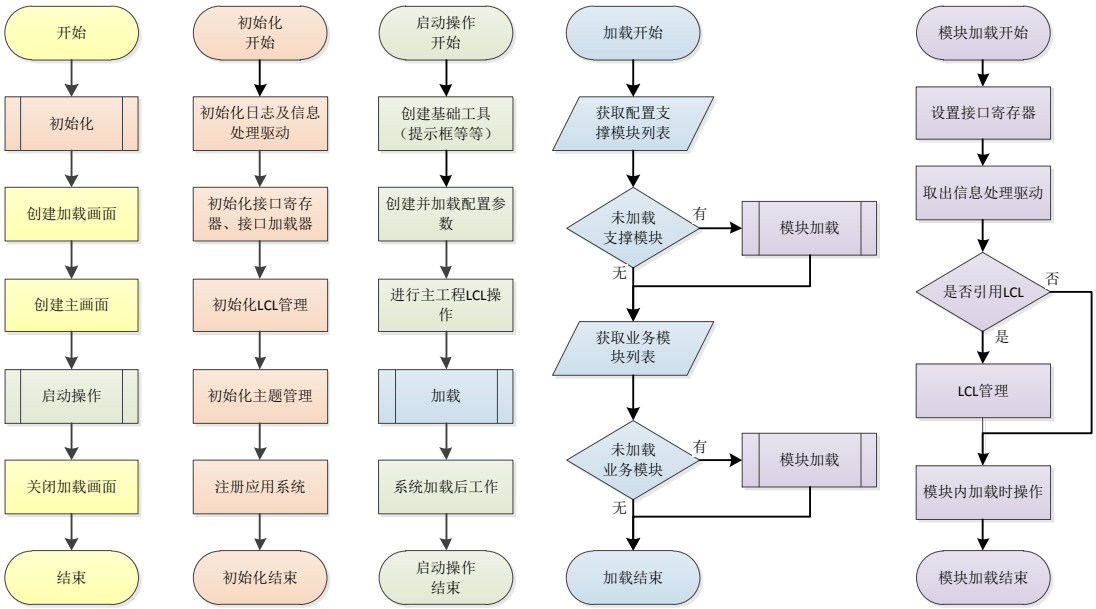
2.2.2 轻量接口通讯机制

系统是以业务边界划分的，按照业务目标去构建小的模块。轻量接口通讯机制提供完整的接口寄存、加载机制，各模块需要相应的服务模块时则尝试获取想的服务接口，在实现完全的自制后注册自身的服务。如此就能降低系统间的依赖性，减少通信成本。



2.4 启动加载流程

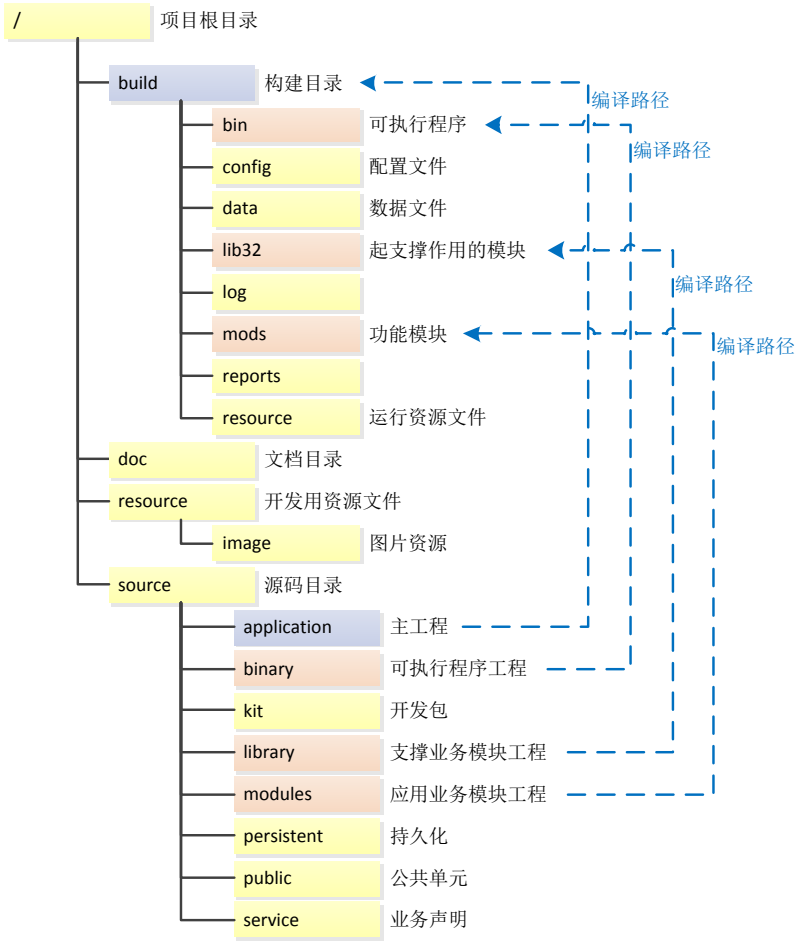
平台启动加载流程如下图所示。


开始
初始化
创建加载画面
创建主画面
启动操作
关闭加载画面
结束
初始化
开始
初始化日志及信息处理驱动
初始化接口寄存器、接口加载器
初始化LCL管理
初始化主题管理
注册应用系统
初始化结束
启动操作
开始
创建基础工具（提示框等等）
创建并加载配置参数
进行主工程LCL操作
加载
系统加载后工作
启动操作
结束
加载开始
获取配置支撑模块列表
未加载支撑模块
有
模块加载
无
获取业务模块列表
未加载业务模块
有
模块加载
无
加载结束
模块加载开始
设置接口寄存器
取出信息处理驱动
是否引用LCL
否
是
LCL管理
模块内加载时操作
模块加载结束

3 平台说明

3.1 目录说明

平台目录结构如下图所示。



3.1.1 源码目录

源码目录位于根目录下 source 目录下。application 是主工程目录，存放主工程的实现；binary 是可执行程序的工程目录，工程的编译目标就是可直接运行的；kit 是与平台无关的公用开发包目录；library 是支撑业务模块工程目标，比如数据服务模块；modules 是应用业务模块工作目录；persistent 是持久化目录，存放持久对象和数据访问对象；public 存放系统的公共单元；service 存放系统的公开业务声明。

3.1.2 构建目录

/bin：可直接运行的文件，里面提供了一些工具，一些命令，供开发或者运行程序时调用。

/lib：程序运行时的基本支撑业务模块

/mods：应用业务模块

3.1.3 其他目录

在根目录下 doc 是项目的文档目录

开发文档应与项目目录一一对应。

3.2 系统核心说明

3.2.1 信息处理/日志

信息处理用于记录特定系统或应用程序组件的传递信息。平台的信息处理的实现方式是采用晨梦工作室的 messager 体系（参见 cm_messager.pas 单元），这是仿自于 JDK 提供的日志框架包 java.util.logging。可以参考 JDK 的 API 文档作更深的了解，这里先作一下简单的说明。

主要包括如下几个部件：

ICMMessager：信息记录对象，对应 JDK 的 Logger。用于记录输出信息。

ICMMessageHandler：用于处理信息的输出，对应 JDK 的 Handler。可以决定信息是输出的方式。

ICMMessageFilter：用于过滤信息，对应 JDK 的 Filter 。可以根据信息级别或者某种条件来决定是否输出该信息。这样达到去除冗余信息的目的。

ICMMessageFormatter：用于格式化信息，对应 JDK 的 Formatter。可以将信息文本格式化成指定的格式。

TEventTypeLevel：用于表示信息的级别。 有如下级别：etlAll(记录所有信息), etlCustom（最低值）, etlDebug, etlInfo, etlWarning, etlError（最高值）,etlOff(不记录任何级别信息)。

对于系统而言，ICMMessager 实例首先会判断信息的级别是否满足输出级别的要求，然后将满足级别要求的信息交给所配置的 ICMMessageHandler 实例来处理，如果信息实例配置了一个 ICMMessageFilter 实例，那么 ICMMessageFilter 实例将会对信息做一次过滤。 ICMMessageHandler 实例接受到信息后，根据其所配置的格式化实例 ICMMessageFormatter 来改变信息的格式，根据所配置的 ICMMessageFilter 实例和 TEventTypeLevel 值来再次过滤信息，最后输出到该 ICMMessageHandler 实例所指定的输出位置中，该输出位置可以是控制台，文件，网络 socket 甚至是内存缓冲区。

同时信息处理也不失其灵活性，你可以定制自己所需要的 ICMMessageHandler，将信息按照自定义的需求输出到不同的位置，同时 ICMMessageFilter、ICMMessageFormatter 都可以自定义扩展实现。

平台默认的 ICMMessageHandler 是一个日志的形式的实现。所以在未指定 ICMMessageHandler 时，信息都是进行日志输出。

以下是一个查找指定名字的 Messager 的例子。

```
procedure TTest.test;
var
  messager: TCMMessager;
begin
  messager := TCMMessageManager.GetInstance.GetMessager('test');
  messager.Info('hello world');
end;
```

也可能通过继承 TCMMessageable、TCMMessageableComponent 等直接获取信息处理的能力。

```
type

  { TTest }

  TTest = class(TCMMessageable)
  public
    procedure test;
  end;

implementation

{ TTest }

begin
  Messager.Info('hello world');
end;
```

3.2.2 接口寄存器

接口寄存器是平台的轻量接口通讯机制。

3.2.3 接口加载器

接口加载器提供三种加载模块库文件的方法。

3.3 系统支撑说明

3.3.1 参数

参数配置应具有良好的结构性和扩展性，平台的配置参数使用可扩展标记语言 XML 进行参数配置。数据库参数因为依赖于数据库服务，所以应作为一种业务存在，这里不作说明。下面通过几个问题的方式来对配置参数进行说明。

参数在什么时候进行加载？配置参数在平台启动时加载。

参数在哪里进行配置？加载的配置文件置于构建目录下 config 目录下。预先加载的是常量（详见/source/public/uConstant.pas）定义的 DefaultConfigFileName 所指定的文件（默认是 config/mpos.xml），你可以在其下增加你所需要的参数。也可以单独撰写配置文件，将文件路径名加入到 configFiles 元素节点下。

```
<?xml version="1.0" encoding="UTF-8"?>
<mpos>
    <resources>
        <ico>resource/logo.ico</ico>
        <logo>resource/logo.png</logo>
    </resources>
    <configFiles>
         <navigator>config/navigator.xml</navigator>
         <test>config/test.xml</test>
    </configFiles>
</mpos>
```

怎样对参数进行调用？所有配置参数加载在一个 ICMParameter 实例中，你可在接口寄存器中获得这一实例，也可以在公共系统中直接访问这一实例。调用 Get 方法获得参数数据，下面给出一组使用举例。

```
<test>
    <!-- 参数名：name；参数值：value-->
    <name>value</name>

    <!-- 参数名：myName；参数值：myValue-->
    <name name="myName">123</name>

    <name2>
        <desc>hello world</desc>
    </name2>
</test>
```

```
procedure TTest.test;
begin
  // 获得“value”
  AppSystem.GetParameter.Get('test.name').AsString;
  // 获得“123”
  AppSystem.GetParameter.Get('test.myName').AsString;
  // 获得能转换的类型值，否则为默认值
  AppSystem.GetParameter.Get('test.myName').AsInteger;
  // 获得“hello world”
  AppSystem.GetParameter.Get('test.name2.desc').AsString;
  // 与上等效
  AppSystem.GetParameter.Get('test.name2').Get('desc').AsString;
  // 获得“”，不存在的节点获得默认值，可以通过调用IsNull判断是否存在，
  //调用DataType获得数据类型
  AppSystem.GetParameter.Get('test.haha').AsString;
end;
```

配置文件的格式是怎样的？参数的层次是与 XML 的树状节点一一对应的，一般将元素名作为参数名，如果存在属性“name”则以这一属性值作为参数名。参数名必须是字母或下划线开头，且仅包含字母、数字的下划线。文本作为参数值，特殊字符需要转义。

3.3.2 主题

主题的配置文件为常量 ThemeConfigFileName 定义文件（默认是 config/themes.xml）。在根节点下可配置多个 theme 元素节点作为不同的主题配置，其应指定 name 属性作为主题名。下属主题配置可参照参数配置方法。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 主题列表，修改时应修改每一个主题的对应位置  -->
<themes>
    <!-- 第一个主题  -->
    <theme name='lavender' title="薰衣草">
        <mainForm>
            <rightWidth>164</rightWidth>
        </mainForm>

        <!-- 基本的 -->
        <boardColor>$ffffff</boardColor>
        <panelColor>$fbd5dd</panelColor>
        <defaultFont>
            <color>0</color>
            <size>11</size>
            <name>宋体</name>
        </defaultFont>
```

要获得主题能力应实现 IThemeable 接口（参见 cm_theme.pas），并加入主题管理。当系统切换主题时主题管理会通知每一个 IThemeable。

```
procedure TTestThemeable.SetTheme(ATheme: ITheme);
begin
  inherited SetTheme(ATheme);
  PanelClient.Color := ATheme.GetParameter.Get('panelColor').AsInteger;
  PanelRight.Color := PanelClient.Color;
  PanelBottom.Color := ATheme.GetParameter.Get('footer.color').AsInteger;
  if not ATheme.GetParameter.Get('mainForm').IsNull then
    PanelRight.Width := ATheme.GetParameter.Get('mainForm.rightWidth').AsInteger;
end;
```

主题参数的调用以配置主题的节点为基点，调用方式请参照配置参数说明。

3.3.3 应用系统

应用系统功能包含系统的基本信息；错误和日志输出工具；对配置参数的直接访问；对外部定义的属性和环境变量的访问。下图是应用系统的定义。

```
IAppSystem = interface(ICMBase)
  ['{0D16D9B0-C131-4A14-B14A-E55FB2EFE41D}']
  function GetVersion: string;           //系统版本
  function IsTestMode: Boolean;
  function GetStartTime: TDateTime;      //启动时间
  function GetParameter: ICMParameter;   //配置参数
  function GetMsgBar: ICMMsgBar;         //消息显示条
  function GetMsgBox: ICMMsgBox;         //消息显示框
  function GetLog: ICMLog;               //系统日志
  function GetWorkRect: TRect;           //系统在屏幕工作区域
  function GetServiceRect: TRect;        //分配给业务在屏幕的工作区域
  procedure AddSystemListener(l: ISystemListener);  //添加指定的系统侦听器
end;
```

函数 GetMsgBar、GetMsgBox 请尽量只用于系统异常提示。

3.3.4 抽象窗口工具

抽象窗口工具作为平台的一种图形界面方案。

……

3.3.5 ……

3.4 业务拓展说明

业务拓展包含支撑业务层和应用业务层。

3.4.1 支撑业务

支撑业务主要是能够为其他业务模块提供服务起一定支撑作用的模块。例如：数据服务模块（数据库访问）、基本服务（监控、调度、资源管理等）等等。

3.4.2 应用业务

应用业务是一些实现特定业务功能的模块。它们不提供服务功能，但可能与其他应用业务模块间存在协作关系。例如：销售、支付模块等等。

4 开发流程与开发规范

软件开发流程是软件设计思路和方法的一般过程，这里我们仅说明在平台之上的编写接入流程。

4.1 开发原则

业务设计应尽量遵循设计模式的六大原则。使用相应的设计模式是为了更好的代码重用性，可读性，可靠性，可维护性。

业务设计应遵循以下原则

单一职责原则：每个业务只需要实现自己的业务逻辑就可以了，比如订单管理模块，它只需要处理订单的业务逻辑就可以了，其它的不必考虑。

服务自治原则：每个业务从开发、测试、运维等都是独立的，包括存储的数据也应是独立的，自己有一套完整的流程，我们完全可以把它当成一个项目来对待。除支撑模块外不应依赖于其他模块。

轻量级通信原则：即跨业务时使用接口寄存器进行通信。

接口明确原则：由于业务之间可能存在着调用关系，为了尽量避免以后由于某个业务的接口变化而导致其他业务都做调整，在设计之初就要考虑到所有情况，让接口尽量做的更通用，更灵活，从而尽量避免其他模块也做调整。

4.2 业务开发流程

下面以一个 hello world 业务的接入说明这个流程。

4.2.1 新建一个业务模块

下面是一个空白业务的模板。引用单元：Classes、cm_InterfaceRegister、cm_InterfaceLoader 、cm_Plat 和引出函数：LoadExport、UnloadExport 是固定的

```
library test;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}cthreads,{$ENDIF}
  Classes, cm_InterfaceRegister, cm_InterfaceLoader, cm_Plat
  { you can add units after this / 你可以在这之后增加单元 };

function LoadExport(ARegister: ICMInterfaceRegister;
                    const AInfo: ICMLibInfo): Boolean; stdcall;
begin
  Result := False;
  if not SetInterfaceRegister(ARegister) then
    Exit;
  {-------------------------------------------
  这里增加加载库时代码
  -------------------------------------------}
  Result := True;
end;
```

```
function UnloadExport(AStatus: Integer): Boolean; stdcall;
begin
  Result := False;
  {-----------------------------------------
  这里增加卸载库时代码
  -----------------------------------------}
  Result := True;
end;

exports
  LoadExport, UnloadExport;

begin
end.
```

4.2.2 声明业务

作为一个独立的业务模块一般存在业务的引出，这就需要对业务进行声明。声明文件的位置应该是所需调用这个业务的一个共同访问的位置。

```
type

  { ITest
    // 测试业务。
  }

  ITest = interface(ICMBase)
    ['{B9CBE1BF-2E9B-4F39-921B-56A174ADAE3D}']
    procedure Test;
  end;
```

4.2.3 实现业务

引用业务声明单元，进行业务实现。

```
uses
  Classes, SysUtils, cm_messager,
  uTest;

type

  { THelloWorld }

  THelloWorld = class(TCMMessageable, ITest)
  public
    procedure Test;
  end;

implementation

{ THelloWorld }

procedure THelloWorld.Test;
begin
  Messager.Info('hello world');
end;
```

4.2.4 引出业务

增加业务声明与实现单元。

```
uses
  {$IFDEF UNIX}cthreads,{$ENDIF}
  Classes, cm_InterfaceRegister, cm_InterfaceLoader, cm_Plat
  { you can add units after this / 你可以在这之后增加单元 },
  uTest, uHelloWorld;
```

放入接口寄存器引出业务。

```
function LoadExport(ARegister: ICMInterfaceRegister;
                    const AInfo: ICMLibInfo): Boolean; stdcall;
begin
  Result := False;
  if not SetInterfaceRegister(ARegister) then
    Exit;
  {-------------------------------------------
  这里增加加载库时代码
  -------------------------------------------}
  //引出测试业务
  Result := InterfaceRegister.PutInterface('测试业务',
                                           ITest, THelloWorld.Create) >= 0;
```

4.2.5 使用业务

系统加载完成后，可在任何地方引用 cm_Plat 和业务定义单元，通过其声明的接口寄存器取出相应业务进行使用。

```
var
  test: ITest;
begin
  if InterfaceRegister.OutInterface(ITest, test) then
    test.Test;
end;
```

4.3 业务规范

业务声明与实现分开

一个模块中可以视情况包含多个业务

界面单元和业务处理单元禁止出现任何 SQL 语句。

4.4 其他规范

不建议业务处理的包引用 LCL

持久化要求

业务模块应统一命名前缀

不允许显示创建计时器

声明的全局变量必须以初始值，这个编译器有时不会给你默认值。

5 异常处理与测试

5.1 异常处理

异常可以分为系统异常（如网络突然断开）和业务异常（如用户的输入值超出最大范围），业务异常必须被转化为业务执行的结果。

不同库的异常处理是独立的，所以错误无法抛出包外。所有异常都应在包内处理(即所有对库外引出的方法存在异常可能时，都应尝试捕获，并且不能抛出)，可以通过系统提示工具进行友好提示，或转换为业务结果。

数据访问层不得向上层隐藏任何异常（受限于编译器，这个异常只能传出而不能抛出）。

建议在模块内明确区分业务执行的结果和系统异常。比如验证用户的合法性，如果对应的用户 ID 不存在，不应该抛出异常，而是返回（或通过 out 参数）一个表示验证结果的枚举值，这属于业务执行的结果。但是，如果在从数据库中提取用户信息时，数据库连接突然断开，则应该抛出系统异常。

表现层应是友好地呈现错误信息。

5.2 测试方法

对外公布的方法（过程、函数）应在方法体开始和结尾输出调试日志。方便进行测试。

```
procedure THelloWorld.Test;
begin
  Messager.Debug('Test()...');
  // Do something you need to do.

  Messager.Debug('Test().');
end;
```

模块测试

6 附

6.1 开发工具的局限性

string 作为基本类型

数据类型

作为编译型语言，类型信息在编译时就已经确定。在运行时如果不是同一编译文件中的类型，相同声明的类型是不可能相等的。当然可以对其强制转换后作用，但无法判断他们的类型信息，同时如果有一个编译文件类型是在修改后编译的，强制转换后使用时内存结构不一致导致宕机。

6.2 与图形界面说

6.2.1 为什么有图形界面的问题

6.2.2 开发工具现状

6.2.3 解决方案

6.3 常见问答

6.2.1 为什么要面向接口编程

6.2.2 接口与抽象类有什么区别

参考文献

[1] 泊川．面向接口编程优缺点[EB/OL]．https://blog.csdn.net/wantken/article/details/31763669，2018-10-27．

[1] 有梦就能实现．软件各种架构图收集[EB/OL]．https://www.cnblogs.com/firstdream/p/7145481.html，2018-10-27．

[2] 刘腾红，刘婧珏．信息系统分析与设计[M]．北京:清华大学出版社，2010.9．

法律声明

本手册仅限公司内部交流、研究使用，禁止外用。