插件化开发

动态导入

运行时，根据用户需求（提供字符串），找到模块的资源动态加载起来。

1、`内建函数__import__()`

`__import__(name, globals=None, locals=None, fromlist=(), level=0)`
name，模块名
import语句本质上就是调用这个函数。但是不鼓励直接使用它。建议使用`importlib.import_module()`。
`sys = __import__('sys')`等价于 import sys

```
# test2.py
class A:
    def showme(self):
        print('I am A')

# 主程序模块test.py
if __name__ == "__main__":
    mod = __import__('test1')
    cls = getattr(mod,'A')
    cls().showme()
```

2、importlib.import_module()

`importlib.import_module(name, package=None)`
支持绝对导入和相对导入，如果是相对导入，package必须设置。

```
# test2.py
class A:
    def showme(self):
        print('I am A')

# 主程序模块test.py
import importlib

def plugin_load(plugin_name:str, sep=":"):
    m, _, c = plugin_name.partition(sep)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)
    return cls()

if __name__ == '__main__':
    # 装载插件
    a = plugin_load('test1:A')
    a.showme()
```

上面的例子就是插件化编程的核心代码。

插件化编程技术

依赖的技术

反射：运行时获取类型的信息，可以动态维护类型数据
动态import：推荐使用importlib模块，实现动态import模块的能力
多线程：可以开启一个线程，等待用户输入，从而加载指定名称的模块

加载的时机

什么时候加载合适?
程序启动的时候，还是程序运行中?

1. 程序启动时
   像pycharm这样的工具，需要很多组件，这些组件也可能是插件，启动的时候扫描固定的目录，加载插件。
2. 程序运行中
   程序运行过程中，接受用户指令或请求，启动相应的插件

两种方式各有利弊，如果插件过多，会导致程序启动很慢，如果用户需要时再加载，如果插件太大或者依赖多，插件也会启动慢。
所以先加载必须的、常用的插件，其他插件使用时，再动态载入。

应用

软件的设计不可能尽善尽美，或者在某些功能上，不可能做的专业，需要专业的客户自己增强。比如Photoshop的滤镜插件。

Notepad++，它只需要做好一个文本编辑器就可以了，其它增强功能都通过插件的方式提供。
拼写检查、HTML预览、正则插件等。
要定义规范，定义插件从哪里加载、如何加载、必须实现的功能等。

接口和插件的区别?
接口往往是暴露出来的功能，例如模块提供的函数或方法，加载模块后调用这些函数完成功能。接口也是一种规范，它约定了必须实现的功能（必须提供某名称的函数），但是不关心怎么实现这个功能。
插件是把模块加载到系统中，运行它，增强当前系统功能，或者提供系统不具备的功能，往往插件技术应用在框架设计中。系统本身设计简单化、轻量级，实现基本功能后，其他功能通过插件加入进来，方便扩展。