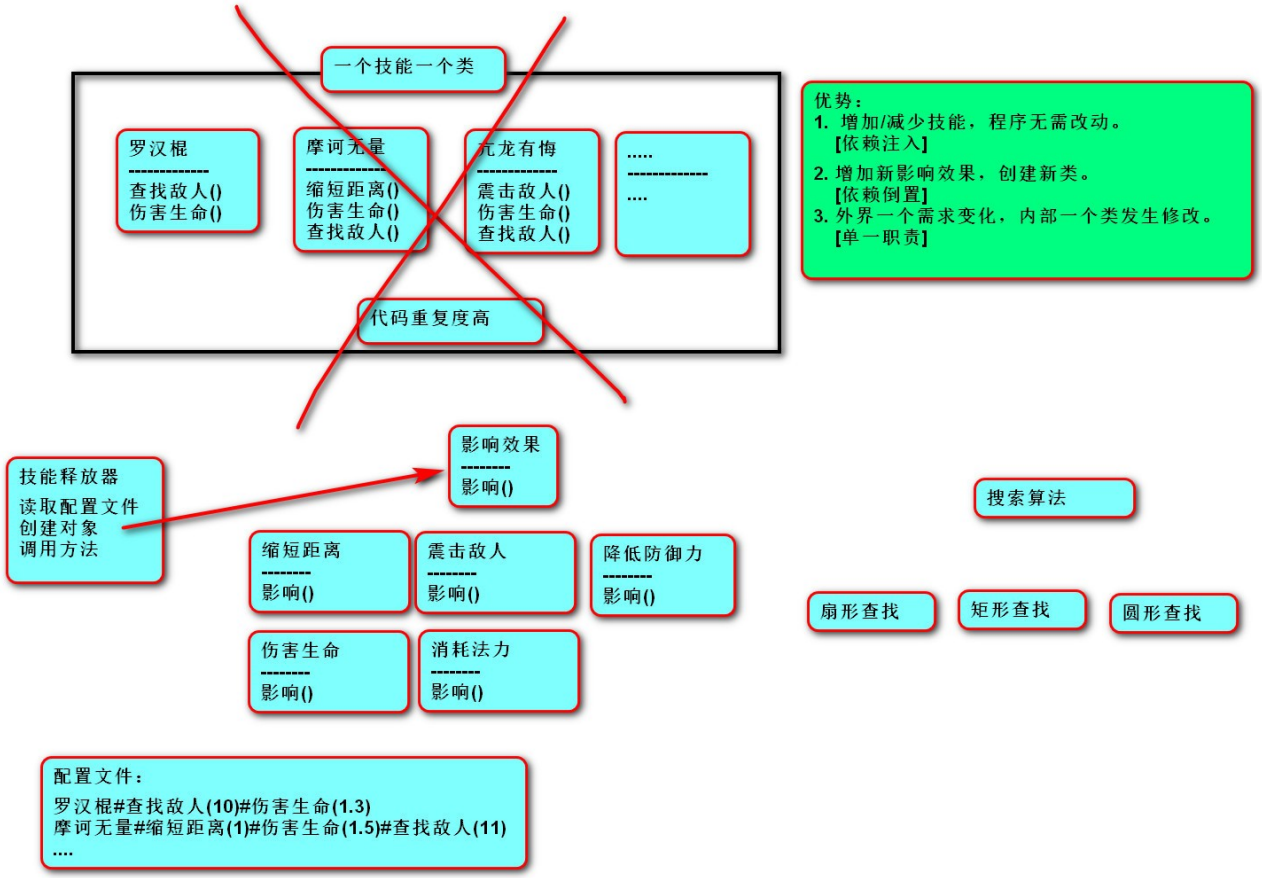

"""

```
    技能系统
    11:05
"""
class SkillImpactEffect:
    """
        技能影响效果
    """
    def impact(self):
        pass
class DamageEffect(SkillImpactEffect):
    """
        伤害效果
    """
    def __init__(self, value=0):
        self.value = value
    def impact(self):
        print("扣你%d血" % self.value)
class LowerDeffenseEffect(SkillImpactEffect):
    """
        降低防御力效果
```

```
    """
    def __init__(self, ratio=0, time=0):
        self.ratio = ratio
        self.time = time
    def impact(self):
        print("降低%.1f防御力%d秒" % (self.ratio,
self.time))
class CostSPEffect(SkillImpactEffect):
    """
        消耗法力效果
    """
    def __init__(self, value=0):
        self.value = value
    def impact(self):
        print("消耗%d法力" % self.value)
class SkillDeployer:
    """
        技能释放器
    """
    def __init__(self, name=""):
        self.name = name
        # 读取配置文件
        self.__dict_skill_config =
self.__locad_config_file()
        # 创建对象
        self.__list_effect_object =
self.__create_effect_object()
    def __locad_config_file(self):
        # 读取代码....(略)
        return {
            "降龙十八掌": ["DamageEffect(500)",
"CostSPEffect(50)"],
            "乾坤大挪移": ["DamageEffect(500)",
"LowerDeffenseEffect(0.3,10)"],
        }
    def __create_effect_object(self):
        # 在字典中,根据当前技能名称,找出需要的影响效果名称.
        # 降龙十八掌 --> ["DamageEffect(500)",
"CostSPEffect(50)"]
        list_effect_name =
self.__dict_skill_config[self.name]
```

```
        # list_effect_object = []
        # for item in list_effect_name:
        #     # 字符串"DamageEffect(500)" --> 对象DamageEffect(500)
        #     effect_object = eval(item)
        #     list_effect_object.append(effect_object)
        # return list_effect_object
        return [eval(item) for item in list_effect_name]
    def generate_skill(self):
        """
            生成技能 -- 执行当前技能的影响效果
        """
        # 调用方法
        print(self.name, "技能释放啦")
        for item in self.__list_effect_object:
            item.impact()
# 测试
skill01 = SkillDeployer("降龙十八掌")
skill01.generate_skill()
skill02 = SkillDeployer("乾坤大挪移")
skill02.generate_skill()
```

```
"""
    面向对象三大特征:
        封装:
            语法:
                数据:一个类包装多个变量.
                行为:隐藏细节,注意必要.
            设计:分
        继承:
            语法:子类拥有父类成员
            设计:隔
        多态:
            语法:重写(子覆盖父)
            做法:重写 + 创建子类对象 --> 调用父执行子(目标)
            设计:干
                def 函数(爸爸):
                    爸爸.功能()
```

```
            函数(儿子())
    类与类关系:
        泛化(继承)/组合(实例变量)/依赖(参数)
    面向对象设计原则:
        开闭原则:允许增,不能改.
        单一职责:一个变化点
        依赖倒置:调用父,而不调用子.
        组合复用:通过变量(参数/实例变量)调用,而不是通过继承调用.
        里氏替换:扩展重写
        迪米特法则:低耦合
"""
```

模块 Module

定义

包含一系列数据、函数、类的文件，通常以.py 结尾。

作用

让一些相关的数据，函数，类有逻辑的组织在一起，使逻辑结构更加清晰。
有利于多人合作开发。

导入

import

1. 语法：
   import 模块名
   import 模块名 as 别名
2. 作用：将某模块整体导入到当前模块中

3. 使用：模块名.成员

from import

1. 语法：
   from 模块名 import 成员名[as 别名 1]
   作用：将模块内的一个或多个成员导入到当前模块的作用域中。

from import *

1. 语法：from 模块名 import *
2. 作用：将某模块的所有成员导入到当前模块。
3. 模块中以下划线(_)开头的属性，不会被导入，通常称这些成员为隐藏成员。

```
"""
    导入模块
        练习1:exercise01.py
        练习2:
            将学生管理系统,分为4个模块.
            model.py    ---> XXXModel
                数据模型
            bll.py    ---> XXXController
                业务逻辑层 business  logic  layer
            usl.py ---> XXXView
                界面处理层 user show layer
            main.py ---> 调用XXXView的代码
                程序入口
        15:30
"""
# 方式1
# 语法:import 模块
# 本质:使用变量module01存储该模块地址
# 优点:不用担心成员冲突问题
# 使用:需要通过变量访问
import module01
module01.fun01()
import module01 as m
m.fun01()
# 方式2
# 本质:将模块指定成员导入到当前模块作用域中
# from 模块 import 成员
# 使用:直接使用导入的成员
# 缺点:可能造成成员冲突
```

```
from module01 import fun01
from module01 import MyClass01
def fun01():
   print("demo01--fun01")
fun01()
c01 = MyClass01()
c01.fun02()
# 方式3
# from 模块 import *
# 优点:可以一次导入多个成员,避免一个一个导入.
# 缺点: 代码可读性不高,命名冲突几率更大.
from module01 import *
# from module02 import *
# from module03 import *
fun01()
c01 = MyClass01()
c01.fun02()

"""
   模块相关概念
"""
from module05 import *
fun01()
# 1. 隐藏模块成员
# _fun02()# 不能访问隐藏成员
# fun03()# 不能访问__all__以外的成员
# from module05 import _fun02
# _fun02()# 能访问隐藏成员
# from module05 import fun03
# fun03() # 能访问__all__以外的成员
# 2. 查看当前模块的文档注释
print(__doc__)
# 3. 查看当前模块的文件路径
# /home/tarena/1906/month01/code/day14/demo03.py
print(__file__)
# 4.查看当前模块名称 --- 判断当前模块是否为主模块
if __name__ == "__main__":
   print("是主模块(程序从当前模块开始执行)")
from package01.package02 import *
module03.fun01()
```

4.

模块变量

__all__变量：定义可导出成员，仅对 from xx import *语句有效。
__doc__变量：文档字符串。
__file__变量：模块对应的文件路径名。
__name__变量：模块自身名字，可以判断是否为主模块。
当此模块作为主模块(第一个运行的模块)运行时，__name__绑定'__main__'，不是主模块，而是被其它模块导入时,存储模块名。

加载过程

在模块导入时，模块的所有语句会执行。
如果一个模块已经导入，则再次导入时不会重新执行模块内的语句。

分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。
2. 标准库模块，安装 Python 时已安装且可直接使用。
3. 第三方模块（通常为开源），需要自己安装。
4. 用户自己编写的模块（可以作为其他人的第三方模块）

搜索顺序

搜索内建模块(builtins)
sys.path 提供的路径，通常第一个是程序运行时的路径。

包 package

定义

将模块以文件夹的形式进行分组管理。

作用

让一些相关的模块组织在一起，使逻辑结构更加清晰。

导入

```
import 包名 [as 包别名]  需要设置__all__
import  包名.模块名 [as 模块新名]
import  包名.子包名.模块名 [as 模块新名]

from 包名 import 模块名 [as 模块新名]
from 包名.子包名 import 模块名 [as 模块新名]
from 包名.子包名.模块名 import 成员名 [as 属性新名]

# 导入包内的所有子包和模块
from 包名 import *
from 包名.模块名 import *
```

搜索顺序

sys.path 提供的路径

__init__.py 文件

是包内必须存在的文件
会在包加载时被自动调用

__all__

记录 from 包 import * 语句需要导入的模块

案例：

```
my_ project /
  __init__.py
    main.py
    common/
      __init__.py
        double_list_helper.py
        list_helper.py
  skill_system/
    __init__.py
    skill_deployer.py
        skill_manager.py
"""
    包
python 程序结构
项目(文件夹)
    包
        模块
            类
                函数/方法
```

```
                语句
   17:00
"""
# 需求:导入module03.py
# 方式1
import package01.package02.module03 as m3
m3.fun01()
# 方式2
# from package01.package02.module03 import fun01
# fun01()
```