开发流程与经验分享

开发流程

- 需求分析，技术选型
- 产品设计原型并给出设计稿
- 产品与前后端讨论原型与技术难点
- 迭代原型
- 后端根据原型分解出对象并给出接口设计文档
- 使用teambition创建排期

开发流程

- 前后端同步开发，并将阶段性代码不断上传至测试服务器联调
- 不断迭代原型
- 测试
- 上线

让阅读代码比编写更方便

好名字，好开始~

可能比下顿吃什么还难T T

举个栗子：当前日期

d

date

todays_date

current

current_date

x

变量名传递的信息

- 变量的内容（它代表什么）
- 数据的种类（具名常量、简单变量、用户自定义类型或类）
- 变量的作用域（公开的、私有的、受保护的、类的、包的、全局的作用域）

好名字表达了what

而不是how

员工数据：input_record　OR　employee_data

变量名中的计算值限定词放在末尾

total、sum、average、max、min、record、string等

优点：

1. 突出变量名中最重要的部分
2. 工整，易于阅读

score_sum	scoreSum
score_average	scoreAverage
score_min	scoreMin
score_max	scoreMax

为状态变量命名

- 标记的名字中不应该含有flag。
- 标记应该用枚举类型、具名常量或用作具名常量的全局变量来对其赋值。

```
if flag:
if printFlag == 16:
if computeFlag == 0:
```

```
if data_ready:
if report_type == REPORT_TYPR_ANNUAL:
if not recalculate_need:
```

```
data_ready = True
report_type = REPORT_TYPR_ANNUAL
recalculate_need = False
```

布尔变量命名

- done，表示某件事已经完成
- error，表示有错误发生
- found，表示某个值已经找到
- success或ok，表示一项操作是否成功
- 使用肯定的布尔变量名。if not not_found

应该避免的名字

- 避免使用令人误解的名字或缩写
- 避免使用具有相似含义的名字，比如input和input_value。同一段代码里很容易混淆他们
- 避免使用具有不同含义但却又相似名字的变量。如client_recs和client_reps，至少应使用client_records和client_reports
- 避免使用发音相似的名字，如wrap和rap，讨论代码时会产生麻烦

- 避免在名字中使用数字。如果名字中的数字很重要，请使用数组。如果数组不合适，那么数字就更不合适

- 避免在名字中拼错单词。比如highlight拼错为hilite，那么这个代表highlite?hilite?hilight?

- 不要紧靠大小写区分变量名

- 不要使用与变量含义无关的名字。比如女朋友、男朋友的名字、最爱的啤酒名字。除非程序真与你男女朋友、啤酒有关。但是你也应该明智的认识到这每一项都可能会变，应该用boyfriend、girlfriend、favorite_beer命名更好

- 避免在名字中包含易混淆的字符。如

eyeChart1 eyeChartl eyechartl

hard2Read hardZRead hard2Read

这些包括【数字1、小写L l、大写i I】、【.和，】、【数字0和大写字母O】、【数字2和大写字符Z】

函数的名字

- 函数名字要对返回值进行描述
  - abs()
  - OrderSchema().dump(order)
- 函数名一般用动词+宾语的形式
  - create_password()

使用对仗词

- add / remove
- begin / end
- increment / decrement
- next / previous
- source / target

變量

变量

- 在靠近变量第一次使用的位置初始化
- 注意计数器和累加器。在下一次使用时要重新初始化
- 检查输入参数的合法性
- 减少变量跨度和生存时间
- 开始时使用最严格的的可见性。后期根据需要拓展。将全局变量切换为类变量是十分困难的。

变量

- 每个变量应只有单一用途。两个场景下使用同一个“临时”变量（temp）会使本无关联的两者看似彼此相关。
- 避免混合耦合。如student_amount表示学生数，但当student_amount = -1时表示发生错误。
- 避免使用神秘数值，代码只允许硬编码0和1

抽象，封装，内聚

功能，一目了然

```
uuid_crc32 = crc32(uuid.encode())
try:
    order = db.session.query(Order).filter(
        Order.uuid_crc32 == uuid_crc32,
        Order.uuid == uuid,
        Order.account_id == account_id,
    ).one()
except MultipleResultsFound:
    raise ServerError(message="查询到多个结果，数据库异常")
except NoResultFound:
    raise ValidationError(message="没有查询到结果")
```

OR

```
order = Order.objects.get(uuid, account_id)
```

注释：提升可读性
还是烟雾弹?

To Comment or Not Comment

注释写的糟糕很容易，
写的出色很难。

注释种类

- 重复代码
- 解释代码
- 标记代码（TODO）
- 概述代码
- 代码意图说明，只出要解决的问题，而不是解决方案
- 传达代码无法表达的信息（版权、保密要求、版本号、优化标注、参考文档、设计文档连接、架构文件索引）

注释占用太多时间的原因

- 注释风格过于复杂或耗时，换一种简单的风格
- 难以说明程序干什么，或写注释太难。（没有真正理解程序的信号）

高效注释

- 采用不会打断或抑制修改的注释风格
- 用伪代码编程法减少注释时间
- 将注释集成到你的开发风格中
- 性能不是逃避注释的好借口
- 不要随意添加无关注释

注释技术

注释单行

对于好的代码，很少需要注释单条语句。要注释一行代码，一般有以下两个理由

- 该行代码太复杂，因而需要注释
- 该行语句出过错，你想记下这个错

行尾注释

行尾注释如：

author_name = “Lance”　# 作者姓名

行尾注释问题

- 如果没有整齐排布，会影响代码的视觉结构
- 很难格式化，可能工作的一大段时间用于格式化
- 维护困难，如果所注释的代码变长了，会把注释挤出屏幕外。
- 预留的空间小，注释就要短，可能会不清晰。

行尾注释原则

- 不要对单行代码做行尾注释
- 不要对多行代码做行尾注释，很难说明注释了多少行。如：

```
if data_ready:  # 更新数据

    recharge.is_succeeded = True

    charge.is_paid = True

    ...
```

行尾注释原则

- 行尾注释用于数据声明
- 避免用行尾注释存放维护标记
- 用行尾注释标记块尾，如在多层for嵌套中

注释主代码块

- 注释应标明代码块意图，将没有包含在代码中信息暴露
- 代码本身应尽力做好自说明。
- 将注释转换为子程序，用代码表意。
- 注释代码段时应注重why而不是how

注释主代码块

- 用注释为后面的内容做铺垫
- 让每个注释都有用
- 说明非常规做法
- 别用缩略语，不要有歧义
- 将主次注释区分开，如用tab缩进

注释主代码块

- 错误或语言环境独特点都要加注释。比如使用第三方库有问题，要对某个特殊值做特殊处理，要注释并标注出第三方库名字 当前使用版本 错误原因等
- 给出违背良好编程风格的理由
- 不要注释投机取巧的代码，应重写这部分代码

注释数据声明

- 注释数据单位
- 注释对数值的允许范围
- 注释编码的含义，更好的方法是使用枚举
- 注释对输入数据的限制
- 注释位标志。如1=大于，0=等于，-1=小于

注释数据声明

- 将与变量有关的注释通过变量名关联起来。搜索变量名时能搜索到注释

- 注释全局变量。指出该数据的目的、为何必须是全局数据

注释子程序

- 子程序上应用一两句话简短说明。如果不能说明设计不足
- 注释接口假设。如传入的students以按学号从小到大排序。
- 对子程序的局限性做注释。如计算精准度等
- 说明子程序的全局效果。如会修改全局变量xxx等
- 记录所用算法的来源

注释类

- 说明该类的设计方法。有些东西是很难通过读代码获得的。需要说明类的设计思路、总体设计方法、曾经考虑过又放弃的设计方法等信息。
- 说明局限性、用法假设等
- 注释类接口。让其他人不看类实现，就知道用法
- 不要在类接口处说明实现细节。注意封装

注释文件

- 说明各个文件的意图和内容
- 在注释块中声明法律通告、版本等信息
- 将文件命名成与其内容相关的名字

注释总结

- 该不该注释是个需要认真对待的问题。差劲的注释只会浪费时间帮倒忙，好的注释才有价值。
- 源代码应当含有程序大部分的关键信息。只要程序在跑，源代码比其他资料都能保持更新，所以使用代码自注释是十分有好处的。
- 好代码本身就是最好的说明。如果代码太糟糕，需要过多的注释，那么首先应当尝试改进代码，直至无须过多的注释为止。

注释总结

- 注释应当说出代码无法说出的东西。如概述、设计用意、参考资料等
- 有的注释风格需要许多的重复性劳动，应当抛弃。改进注释风格。

谢谢！