

# 代码的艺术

共 6 个方面：

[《代码的艺术》目的解读](#)

[代码与艺术之间的关系](#)

[软件工程师不等于码农](#)

[正确认识代码实践方面的问题](#)

[怎么样修炼成为优秀的软件工程师](#)

[《代码的艺术》课程总结](#)

## 《代码的艺术》目的解读

### 1.了解公司与学校写代码的不同

学校：代码要求质量低

公司：代码要求质量必须高（服务用户多）

### 2.消除对于程序员这个职业的误解

消除误解，程序员的出入不仅仅是码农

### 3.建立对软件编程的正确认识

“知”与“行”要合一：我们需要对这件事物有一个正确的认识，才会有正确的行动。

同理，写出好代码的前提，是对软件编程有正确的认识。

### 4.明确作为软件工程师的修炼方向

艺术品，由艺术家创造

代码，由软件工程师创造

新的认识。

## 代码与艺术之间的关系

3 个维度

## 1.代码是可以被称为艺术的

艺术的解读：

艺术就是人类通过借助特殊的物质材料与工具，运用一定的审美能力和技巧，在精神与物质材料、心灵与审美对象的相互作用下，进行的充满激情与活力的创造性劳动，可以说它是一种精神文化的创造行为，是人的意识形态和生产形态的有机结合体。

写代码也恰恰要经历这样的一个过程。



写代码的过程：

物质：计算机系统

工具：设计、编写、编译、调试、测试

编写代码需要的是激情

编写代码是一件非常具有创造性的工作

总结：

代码 = 智慧的结晶

反应了一个团队或一个人的精神

## 2. 艺术可以从不同的角度进行解读、研究与创造

举例：蒙娜丽莎的微笑

观众角度：可能只是在欣赏画中的人物微笑

画家角度：可能就会考虑画画的手法、构图、光线明暗、色彩对比等等方面。

举例：达芬奇的自画像

解释：不仅通过观看，还有人物的个性，背景等等揣摩画作。

总结：在艺术方面，可以站在很多不同的角度进行解读。如果要成为一名创作者，*我们需要的不仅仅是欣赏的能力，更重要的是从多角度进行解读、研究与创造的能力。*

### 3.写代码如同艺术创作

写代码的内涵:

- ①写代码这个过程是一个从**无序到有序**的过程。
- ②写代码需要把现实问题转化为数学模型。在写代码的过程中,我们需要有很好的**模型能力**。
- ③写代码实际是一个**认识的过程**。很多时候,编码的过程也是我们认识未知问题的过程。
- ④在写代码的过程中,我们需要**综合的全方位的能力**。包括把握问题的能力、建立模型的能力、沟通协助的能力、编码执行的能力等等。
- ⑤在写好代码之前,首先需要**建立品位**。品味是指我们首先要知道什么是好的代码,什么是不好的代码。这样我们才能去不断地调整自己的行为,然后去学习,去提高我们的编码能力,写出具有**艺术感**的代码。

### 软件工程师不等于码农

软件工程师不只会写代码,还具备综合的素质

#### 1、技术

技术能力是基础。包括但不限于编码能力、数据结构和算法能力、系统结构知识、操作系统知识、计算机网络知识、分布式系统知识等等。

#### 2、产品

要对产品业务有深刻的理解,需要了解产品交互设计、产品数据统计、产品业务运营等。

#### 3、其他

要了解一些管理知识,需要知道项目是怎么管理的,如何去协调多个人一起去完成一个项目。有一些项目需要具有很强的研究与创新方面的能力。

#### 4、经验

至少有 8-10 年的工作经验

总结:

软件工程师绝对不是一个只会简单编写代码就可以的职业。**软件工程师不等于码农**。

## 正确认识代码实践方面的问题

### 1. 什么是好的代码，好的代码有哪些标准



归纳后：



### 2. 不好的代码主要表现的方面

不好的函数名 例如：my

不好的变量名 例如：a,b,c,j,k,temp

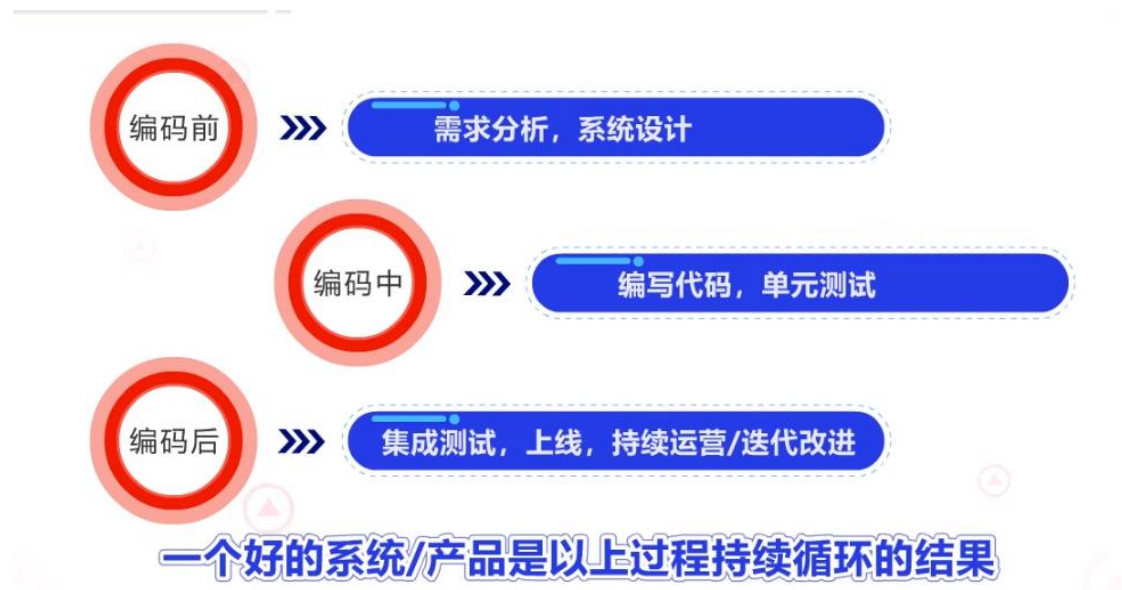
没有主食或注释不清晰

一个函数执行多个功能 例如：LoadFromFileAndCalculate()函数

样式排版 要规避不好的代码样式排版

难以测试的代码 代码没法测试，难写测试用例

### 3.好的代码从哪里来



#### 需求分析和系统设计（重点）

认识需求分析和系统设计的重要性：

前期更多的投入，收益往往最大

清楚需求分析和系统设计的差别：

需求分析主要是定义系统或软件的**黑盒行为**。

即：外部行为。比如，系统从外部来看能够执行什么功能。

系统设计主要是设计系统或软件的**白盒机制**。

即：内部行为。比如，系统从内部来看，是怎么做出来的，为什么这么做。

需求分析的注意要点：

要点一：清楚怎么用寥寥数语勾勒出一个系统的功能。

需求描述的内容基本包括：

- 1.系统类型描述
- 2.系统规模描述
- 3.系统定位和系统差异描述
- 4.系统对外接口功能描述

要点二：需求分析用精确的数据来描述。

需求分析中会涉及大量的数据分析，这些分析都需要精确的数字来进行支撑。

系统设计的注意要点：

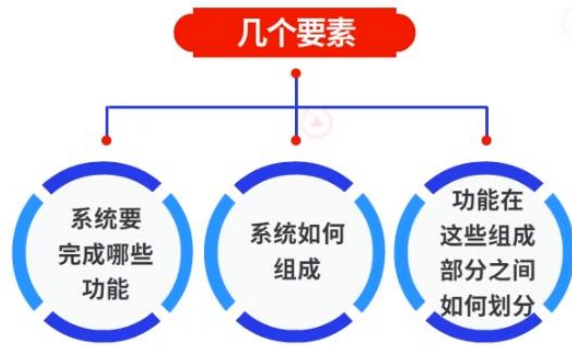
要点一：清楚什么是系统架构

要点二：注意系统设计的约束

要点三：清楚需求是系统设计决策的来源

要点四：系统设计的风格与哲学

要点五：清楚接口的重要性



#### 4 系统设计的注意要点

要点一：清楚什么是系统架构

要点二：注意系统设计的约束

要点三：清楚需求是系统设计决策的来源

要点四：系统设计的风格与哲学

要点五：清楚接口的重要性



要点一：清楚什么是系统架构

要点二：注意系统设计的约束

要点三：清楚需求是系统设计决策的来源

要点四：系统设计的风格与哲学

要点五：清楚接口的重要性





要点一：清楚什么是系统架构

要点二：注意系统设计的约束

要点三：清楚需求是系统设计决策的来源

要点四：系统设计的风格与哲学

要点五：清楚接口的重要性

在同样的需求下，可能出现不同的设计方式



好的系统是在合适假设下的精确平衡

一个通用的系统在某些方面是不如专用系统的

补充解释：

一个好的系统是在合适假设下的**精确平衡**。一个通用的系统在某些方面是不如专用系统的。每个系统每个组件的功能都应该足够的专一和单一。每个组件是指子系统或模块等。功能的单一是复用和扩展的基础。倘若不单一，未来就有可能很难进行复用和扩展。

子系统或模块之间的关系应该是**简单而清晰**的。软件中最复杂的是耦合，如果各系统之间的接口定义非常复杂，那么未来便很难控制系统的健康发展。

值得注意的是，使用全局变量就是在增加系统的耦合，从而增加系统的复杂性，所以在系统中需要**减少使用全局变量**。

减少使用全局变量



#### 4 系统设计的注意要点

要点一：清楚什么是系统架构

要点二：注意系统设计的约束

要点三：清楚需求是系统设计决策的来源

要点四：系统设计的风格与哲学

要点五：清楚接口的重要性

模块对外的函数接口

平台对外的API（很多是RPC或Web API）



系统间通信的协议

系统间存在依赖的数据（比如：给另一个系统提供的词表）

补充：接口重要的原因在于：

接口定义了功能。如果定义的功能不正确，那么系统的可用性与价值便会大打折扣。

接口决定了系统和系统外部之间的关系。相对于内部而言，外部关系确定后非常难以修改。

后期接口修改时主要注意两点：

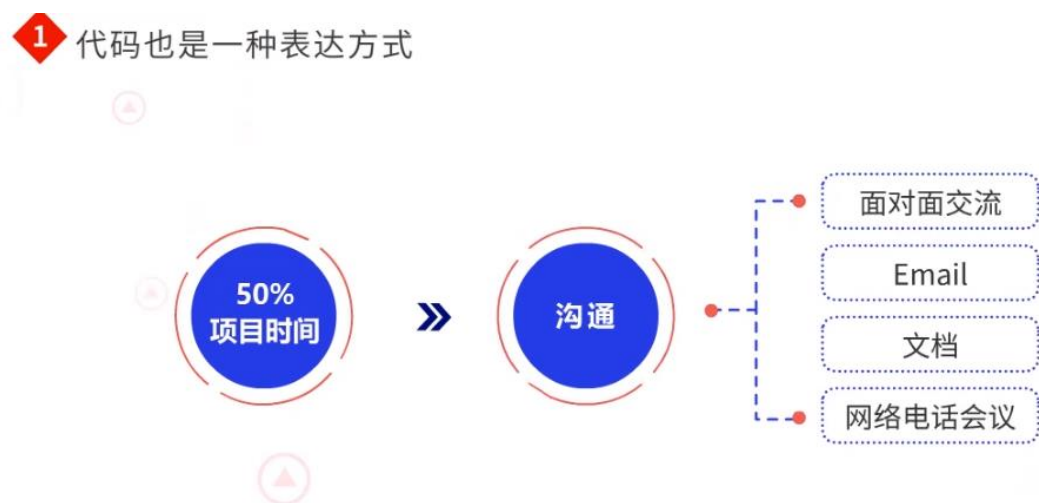
第一，合理好用。新改的接口应该是非常合理好用的。不能使调度方感觉我们做的接口非常难以使用。

第二，修改时需要向前兼容。新改的接口应该尽量实现前项的兼容。不能出现当新接口上线时其他程序无法使用的情况。

如何写好代码：

两个维度：

代码也是一种表达方式：



编程规范主要包含：

如何规范的表达代码。

语言使用的相关注意事项。

基于编程规范，看代码的理想场景是：

看别人的代码，感觉和看自己的代码一样。

看代码时能够专注于逻辑，而不是格式方面。

看代码时不用想太多。

代码书写过程中的细节问题：

9 个方面：

1. 关于模块：

定义：程序的基本组成单位。

需要：

明确的功能。

符合紧内聚，松耦合。

切分模块的方法：

数据类的模块：主要是要完成对数据的封装。封装往往是通过模块内部变量或类的内部变量来实现的。

过程类的模块：本身不含数据。过程类模块可以从文件中去读取一个数据，



或者执行一些相关的操作。过程类模块可以调用其他数据类模块或过程类模块。  
注意事项：

编写程序时，我们需要注意减少模块间的耦合。[减少模块间的耦合](#)，有利于降低软件复杂性，明确接口关系。

## 2. 关于类和函数

类和函数是两种不同的类型，有他们各自适用的范围。另外，遇见和类的成员变量无关的函数时，可以将该函数抽出来，作为一个独立的函数使用，这样便于未来的复用。

## 3. 关于面向对象

本质：数据封装

实际：C 语言是基于对象的，它和 C++ 的区别主要是没有多态和继承

注意：慎地使用多态和继承

## 4. 关于模块内部的组成

文件头：对模块的功能进行简要说明。需要把文件的修改历史写清楚，包括修改时间、修改人和修改内容。在模块内，内容的顺序尽量保持一致，以方便未来对内容的搜索查询

## 5. 关于函数

对于一个函数来说，要有 [明确的单一](#) 的功能

三要素：功能、传入参数和返回值

功能描述是指描述这个函数是做什么的、实现了哪些功能

传入参数描述是指描述这个函数中传入参数的含义和限制条件。

返回值描述是指描述这个函数中返回值都有哪些可能性

函数的规模要足够的短小。

函数的语义：

返回值有 [三种](#) 类型。

第一种类型：在“逻辑判断型”函数中，返回布尔类型的值——True 或 False，表示“真”或“假”。

第二种类型：在“操作型”函数中，作为一个动作，返回成功或失败的结果——SUCCESS 或 ERROR。

第三种类型：在“获取数据型”函数中，返回一个“数据”，或者返回“无数据/获取数据失败”。

推荐：以“单入口、单出口”的方式书写。这种方式能够比较清晰地反映出函数的逻辑

## 6. 关于代码注释

书写注释要做到清晰明确。在编写程序的过程中，先写注释，后写代码

## 7. 关于代码块

思路：把代码中的段落分清楚。文章有段落，代码同样有段落。代码的段落背后表达的是我们对于代码的逻辑理解。包括代码的层次、段落划分、逻辑。代码中的空行或空格是帮助我们表达代码逻辑的，并非可有可无。好的代码可以使人在观看时做过一眼明了。

## 8. 关于命名

命名包括系统命名、子系统命名、模块命名、函数命名、变量命名、常量命名等。

命名重要性：

主要因为：

一是“望名生义”是人的自然反应。不准确的命名会使人产生误导。

二是概念是建立模型的出发点。好的命名是系统设计的基础。

普遍存在的问题：

一是名字中不携带任何信息。

二是名字携带的信息是错误的

#### 9. 关于系统的运营

代码的可监测性

数据收集

## 怎样修炼成为优秀的软件工程师

传统判断维度：

工作时间

代码量

学历

曾就职的公司

重要因素：

#### 1、学习-思考-实践

##### (1) 多学习

软件编写的历史已经超过半个世纪，有太多的经验可以借鉴学习。要不断的学习进步。

##### (2) 多思考

学而不思则罔，思而不学则殆。对于做过的项目要去深入思考，复盘写心得。

##### (3) 多实践

要做到知行合一，我们大部分的心得和成长其实是来自于实践中的经历。在学习和思考的基础之上，要多做项目，把学到的理论运用到真正的工作中。

#### 2、知识-方法-精神

拥有的精神理念：

##### (1) 自由精神、独立思考。

人一定要有自己的思考。不要人云亦云，不要随波逐流。

##### (2) 对完美的不懈追求。

不要做到一定程度就满意了，而是要去不断的追求一个更好的结果。

#### 3、基础知识是根本

掌握的基础得非常全面。

包括数据结构、算法、操作系统、系统结构、计算机网络。包括软件工程、编程思想。

包括逻辑思维能力、归纳总结能力、表达能力。还包括研究能力、分析问题、解决问题的能力等。这些基础的建立，至少也要 5~8 年的时间。

## 《代码的艺术》课程总结

已了解公司与学校写代码又很大不同

已消除对于程序员这个职业的误解

已建立对软件编程的正确认识

已明确作为软件工程师的修炼方向

课程要点总结：

*软件工程师不等于码农*，软件工程师是一个很有深度的职业

代码，我们可以把它写成艺术品。最终成品完全在于我们自身的修养

*不要忘记我们为什么出发*。我们的目的是改变世界，而不是学习编程或者炫耀技术

*好代码的来源不是写好代码*，好代码是一些列工作的结果

*代码是写给别人看的*，别人看不懂的代码就是失败的

*写好代码是有方法的*。系统工程师至少需要 8~10 年的积累。

要能够沉下心来，把基础打好，提升能力。

## Mini-spider 实践课程

两个方面：

[多线程编程](#)

[具体细节处理](#)

### 1. 多线程编程

#### 1. 数据互斥访问

常见错误：将一张表的“添加”和“判断是否存在”分为两个接口。

```
class UrlTable(object):
    def __init__(self):
        self.lock = threading.Lock()
        self.table = {}

    def add(self, url):
        "add url to table"
        ret_val = 'OK'
        self.lock.acquire()
        if url in self.table:
            ret_val = 'ERROR'
        else:
            self.table[url] = True
        self.lock.release()
        return ret_val
```

将添加和判断写进一个函数中

```
class UrlTable(object):
    def __init__(self):
        self.lock = threading.Lock()
        self.table = {}

    def add(self, url):
        self.lock.acquire()
        self.table[url] = True
        self.lock.release()

    def is_in_table(self, url):
        self.lock.acquire()
        ret_val = url in self.table
        self.lock.release()
        return ret_val
```

分别编写添加和判断函数

#### 2. 临界区的注意事项

有锁来保护的区域被称为临界区

示例：

```
class Table(object):
    def __init__(self):
        self.lock = threading.Lock()
        self.table = {}

    def add(self, url, value):
        value = time_consuming_calc()
        self.lock.acquire()
        self.table[url] = value
        self.lock.release()
```

```
class Table(object):
    def __init__(self):
        self.lock = threading.Lock()
        self.table = {}


    def add(self, url, value):
        self.lock.acquire()
        self.table[url] = time_consuming_calc(value)
        self.lock.release()
```

注意：不要把耗费时间的操作放在临界区内执行

### 3.I/O 操作的处理

注意：不能出现无捕捉的 exception

示例：

<pre>class Table(object):     def __init__(self):         self.lock = threading.Lock()         self.table = {}      def add(self, url):         self.lock.acquire()         self.table[url] = file_read(url)         self.lock.release()</pre>		<pre>class Table(object):     def __init__(self):         self.lock = threading.Lock()         self.table = {}      def add(self, url):         self.lock.acquire()         try:             value = file_read(url)         except IOError:             value = None          if value is not None:             self.table[url] = value         self.lock.release()          if value is None:             return 'ERROR'         else:             return 'OK'</pre>	<pre>class Table(object):     def __init__(self):         self.lock = threading.Lock()         self.table = {}      def add(self, url):         try:             value = file_read(url)         except IOError:             return 'ERROR'          self.lock.acquire()         self.table[url] = value         self.lock.release()          return 'OK'</pre>
--	--	---	--

以图中最左边的代码为例，如果不对异常进行捕捉，那么一旦出现问题就不会执行 self.lock.release() 语句，进而导致死锁的发生。

其次，因为异常处理是非常消耗资源的，所以我们也不能像图中中间的代码一样，将异常放在临界区内，要像最右边的代码一样处理。

## 2.具体细节处理

### 1. 种子信息的读取

示例：

```

class Spider(object):
    def __init__(self, feed_file, result_file, max_depth,
                 crawl_interval, crawl_timeout, thread_count):
        ...

        # 添加 seed (url)
        for seed in self._get_seeds():
            if not seed:
                continue
            # 设置深度
            self.url_depth[seed] = 0
            self.crawl_strategy.add_unvisited_url(seed)

    def _get_seeds(self):
        with open(self.feed_file, 'r') as f:
            for line in f.readlines():
                # 删除末尾的'\n'
                yield line.strip()

```

错误：将种子信息读取的逻辑和其他逻辑耦合在一起

模块划分和逻辑的复杂程度是没有关系的

即使是逻辑简单的代码，如果没有做好模块划分，也会变得难于维护

## 2. 程序优雅退出

示例：

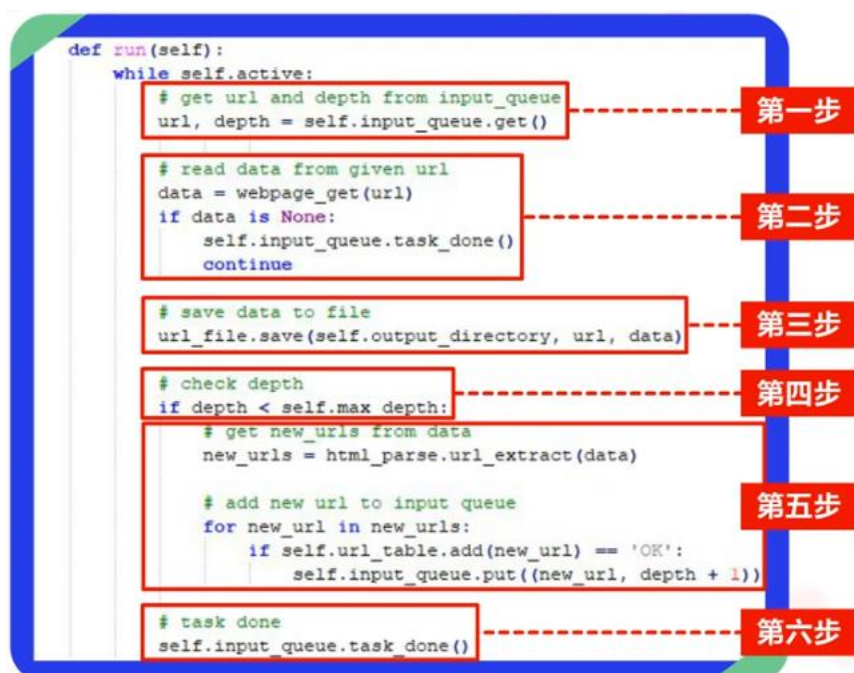
爬虫的逻辑	主程序的逻辑	主程序的逻辑(更好的方式)
<pre> class Crawler(threading.Thread):     ...     def run(self):         while self.active:             # get url and depth from input_queue             url, depth = self.input_queue.get()              # process the task             ...              # task done             self.input_queue.task_done() </pre>	<pre> class MiniSpider(object):     def __init__(self, config, seeds):         ...      def start(self):         # start crawlers         for crawler in self.crawlers:             crawler.start()          self.input_queue.join()      def start():         ...          # initialize spider         spider = MiniSpider(config, seeds)          # start the spider         spider.start() </pre>	<pre> class MiniSpider(object):     def __init__(self, config, seeds):         ...      def start(self):         # start crawlers         for crawler in self.crawlers:             crawler.start()          def wait(self):             self.input_queue.join()      def start():         ...          # initialize spider         spider = MiniSpider(config, seeds)          # start the spider         spider.start()          # wait for finish         spider.wait() </pre>

python 系统库 task\_done(), join()

## 3. 爬虫的主逻辑编码

示例：





好的代码是可以很快理解的，可读性是非常好的  
开发人员在编写代码中需要注意的一个点

## 代码检查规则背景及总体介绍

分为 3 个部分：

1. [代码检查的意义](#)
2. [代码检查场景与工具](#)
3. [代码检查规则与等级](#)

### 1.代码检查的意义

提高代码可读性，统一规范，方便他人维护，长远来看符合公司内部开源战略。  
帮助发现代码缺陷，弥补人工代码评审的疏漏，节省代码评审的时间与成本。  
有助于提前发现问题，节约时间成本，降低缺陷修复成本。  
促进公司编码规范的落地，在规范制定后借助工具进行准入检查。  
提升编码规范的可运营性，针对反馈较多的不合理规范进行调整更新

### 2.代码检查场景与工具

#### 1.代码检查场景

## 1. 代码检查场景



## 2. 代码检查工具与服务



有交互性，共同构成整个代码检查环节

## 3. 代码检查覆盖范围



#### 4.代码检查速度



编码规范：只扫描变更文件，检查代码变更行是否符合规范，速度较快。

缺陷检查：需考虑文件依赖、函数调用关系、代码上下文等，相对耗时。

### 3.代码检查规则分级

规则等级梳理：

1) Error：属于需要强制解决的类型，影响代码合入，应视具体情况不同采取修复、临时豁免、标记误报等措施及时处理。

2) Warning：非强制解决类型，不影响代码含入，很可能存在风险，应尽量修复。

3) Advice：非强制解决类型，级别相对较低，不影响代码含入，可以选择性修复。

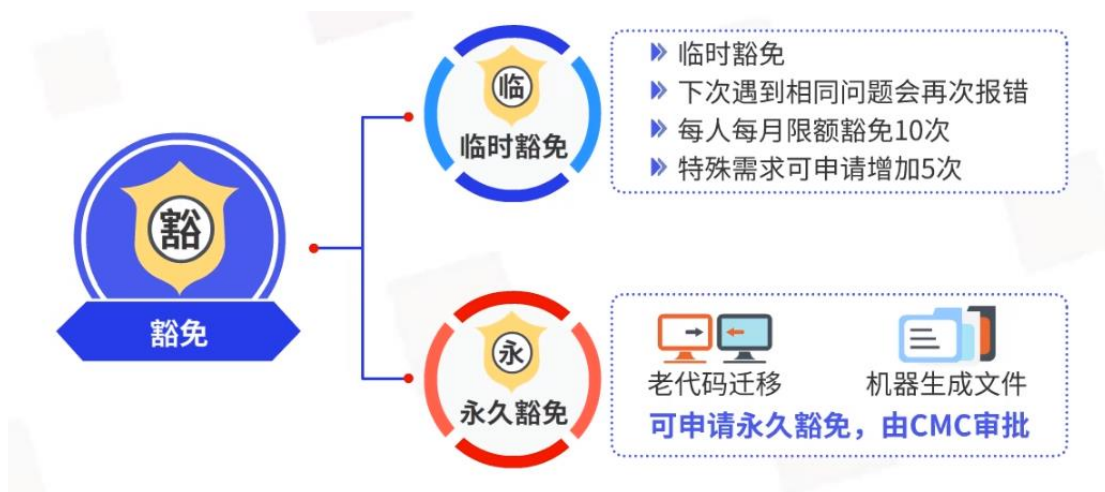
机检任务统一：



评审页间提示:



针对豁免、误报、咨询的说明:





## 代码检查规则：Python 语言案例详解

两个部分：

[1.Python 的代码检查规则](#)

[2.Python 编码惯例](#)

### 1.Python 的代码检查规则



#### 1. Python 代码检查规则的讲解

Python 代码检查规则主要分为四个大类，分别是代码风格规范、引用规范、定义规范和异常处理规范。

代码风格规范（7 类 17 条）：

（1）程序规模规范：

①每行不得超过 120 个字符。



②定义的函数长度不得超过 120 行。

(2) 语句规范

③禁止以分号结束语句。

④在任何情况下，一行只能写一条语句。

(3) 括号使用规范

⑤除非用于明确算术表达式优先级、元组或者隐式行连接，否则尽量避免冗余的括号。

(4) 缩进规范

⑥禁止使用 Tab 进行缩进，而统一使用 4 个空格进行缩进。

需要将单行内容拆成多行写时规定：

⑦与首行保持对齐；或者首行留空，从第二行起统一缩进 4 个空格。

(5) 空行规范

⑧文件级定义（类或全局函数）之间，相隔两个空行；类方法之间，相隔一个空行。

(6) 空格规范

⑨括号之内均不添加空格。

⑩参数列表、索引或切片的左括号前不应加空格。

⑪逗号、分号、冒号之前均不添加空格，而是在它们之后添加一个空格。

⑫所有二元运算符前后各加一个空格。

⑬关键字参数或参数默认值的等号前后不加空格。

(7) 注释规范

⑭每个文件都必须有文件声明，每个文件声明至少必须包括以下三个方面的信息：版权声明、功能和用途介绍、修改人及联系方式

另外在使用文档字符串（docstring）进行注释时，规定：

⑮使用 docstring 描述模块、函数、类和类方法接口时，docstring 必须用三个双引号括起来。

⑯对外接口部分必须使用 docstring 描述，内部接口视情况自行决定是否写 docstring。

⑰接口的 docstring 描述内容至少包括以下三个方面的信息：功能简介、参数、返回值。如果可能抛出异常，必须特别注明。

引用规范：

① 禁止使用 `from...import...` 句式直接导入类或函数，而应在导入库后再行调用。

② 每行只导入一个库。

③ 按标准库、第三方库、应用程序自有库的顺序排列 import，三个部分之间分别留一个空行。

定义规范：

(1) 在变量定义方面，我们有强制的规范规定：

① 局部变量使用全小写字母，单词间使用下划线分隔。

② 定义的全局变量必须写在文件头部。

③ 常量使用全大写字母，单词间使用下划线分隔

(2) 函数的定义规范主要体现在函数的返回值以及默认参数的定义上。

① 函数返回值必须小于或等于 3 个。若返回值大于 3 个，则必须通过各种具名的形式进行包装。

② 仅可使用以下基本类型的常量或字面常量作为默认参数：整数、bool、浮

点数、字符串、None。

(3) 类定义的规范包括了四个方面的内容：

- ① 类的命名使用首字母大写的驼峰式命名法。
- ② 对于类定义的成员：protected 成员使用单下划线前缀；private 成员使用双下划线前缀。
- ③ 如果一个类没有基类，必须继承自 object 类。
- ④ 类构造函数应尽量简单，不能包含可能失败或过于复杂的操作。

异常处理规范：

- ① 禁止使用双参数形式或字符串形式的语法抛出异常。
- ② 如需自定义异常，应在模块内定义名为 Error 的异常基类。并且，该基类必须继承自 Exception。其他异常均由该基类派生而来。
- ③ 除非重新抛出异常，禁止使用 except: 语句捕获所有异常，一般情况下，应使用 except...: 语句捕获具体的异常。
- ④ 捕捉异常时，应当使用 as 语法，禁止使用逗号语法。

## 2. Python 编码惯例

让模块既可被导入又可执行

Python 脚本语言：动态的逐行解释运行，没有统一的程序入口

通常自定义一个 main 函数，并使用一个 if 语句

in 运算符的使用

合理的使用 in 运算符，可以代替大量的重复判断过程，降低时间复杂度，提高代码的运行效率

不使用临时变量交换两个值

无必要引入临时变量来交换两个值。

用序列构建字符串

利用一个空字符串和 join 函数，就可以避免重复，高效完成相应字符串的构建。