售票系统代码风格规范

黎睿德

2017.4.6

Java 代码规范

本 Java 代码规范以 SUN 的标准 Java 代码规范为基础.

1. 标识符命名规范

1.1 概述

标识符的命名力求做到统一、达意和简洁。尽量做到每个人按照规范来，多人开发如一人开发一样。

1.1.1 统一

统一是指，对于同一个概念，在程序中用同一种表示方法，比如对于供应商，既可以用 supplier，也可以用 provider，但是我们只能选定一个使用，至少在一个 Java 项目中保持统一。统一是作为重要的，如果对同一概念有不同的表示方法，会使代码混乱难以理解。即使不能取得好的名称，但是只要统一，阅读起来也不会太困难，因为阅读者只要理解一次。

1.1.2 达意

达意是指，标识符能准确的表达出它所代表的意义，比如： newSupplier, OrderPaymentGatewayService 等；而 supplier1, service2，idtts 等则不是好的命名方式。准确有两成含义，一是正确，而是丰富。如果给一个代表供应商的变量起名是 order，显然没有正确表达。同样的，supplier1, 远没有 targetSupplier 意义丰富。

1.1.3 简洁

简洁是指，在统一和达意的前提下，用尽量少的标识符。如果不能达意，宁愿不要简洁。比如：theOrderNameOfTheTargetSupplierWhichIsTransfered 太长，transferedTargetSupplierOrderName 则较好，但是 transTgtSplOrdNm 就不好了。省略元音的缩写方式不要使用，我们的英语往往还没有好到看得懂奇怪的缩写。

1.1.4 骆驼法则

Java 中，除了包名，静态常量等特殊情况，大部分情况下标识符使用骆驼法则，即单词之间不使用特殊符号分割，而是通过首字母大写来分割。比如: supplierName, addNewContract，而不是 supplier_name, add_new_contract。

1.1.5 英文 vs 拼音

尽量使用通俗易懂的英文单词，如果不会可以向队友求助，实在不行则使用汉语拼音，避免拼音与英文混用。比如表示归档，用 archive 比较好，用 pigeonhole 则不好，用 guiDang 尚可接受。

1.2 包名

使用小写字母如 com.amerisia.ebills，不要 com.amerisia.Ebills
单词间不要用字符隔开，比如 com.amerisia.ebills，而不要 com.amerisia.ebills _util

1.3 类名

1.3.1 首字母大写

类名要首字母大写，比如 LCIssueInfoManagerEJB, LCIssueAction；不要 lcIssueInfoManagerEJB, lcIssueAction.

1.3.2 后缀

类名往往用不同的后缀表达额外的意思，如下表：

后缀名	意义	举例
EJB	表示这个类为 EJB 类	LCIssueInfoManagerEJB
Service	表明这个类是个服务类，里面包含了给其他类提同业务服务的方法	PaymentOrderService
Impl	这个类是一个实现类，而不是接口	PaymentOrderServiceImpl
Inter	这个类是一个接口	LifeCycleInter
Dao	这个类封装了数据访问方法	PaymentOrderDao
Action	直接处理页面请求，管理页面逻辑了类	UpdateOrderListAction
Listen	响应某种事件的类	PaymentSuccessListener

er		
Event	这个类代表了某种事件	PaymentSuccessEvent
Servlet	一个 Servlet	PaymentCallbackServlet
Factory	生成某种对象工厂的类	PaymentOrderFactory
Adapter	用来连接某种以前不被支持的对象的类	DatabaseLogAdapter
Job	某种按时间运行的任务	PaymentOrderCancelJob
Wrapper	这是一个包装类，为了给某个类提供没有的能力	SelectableOrderListWrapper
Bean	这是一个 POJO	MenuStateBean

1.4 方法名

首字母小写，如 addOrder() 不要 AddOrder()
动词在前，如 addOrder()，不要 orderAdd()

查询方法要查询的内容在前，条件在后。 如 getXxByXx（）
动词前缀往往表达特定的含义，如下表：

前缀名	意义	举例
create	创建	createOrder()
delete	删除	deleteOrder()
add	创建，暗示新创建的对象属于某个集合	addPaidOrder()
remove	删除	removeOrder()
init 或则 initialize	初始化，暗示会做些诸如获取资源等特殊动作	initializeObjectPool
destroy	销毁，暗示会做些诸如释放资源的特殊动作	destroyObjectPool
open	打开	openConnection()
close	关闭	closeConnection()<
read	读取	readUserName()
write	写入	writeUserName()

get	获得	getName()
set	设置	setName()
prepare	准备	prepareOrderList()
copy	复制	copyCustomerList()
modity	修改	modifyActualTotalAmount()
calculate	数值计算	calculateCommission()
do	执行某个过程或流程	doOrderCancelJob()
dispatch	判断程序流程转向	dispatchUserRequest()
start	开始	startOrderProcessing()
stop	结束	stopOrderProcessing()
send	发送某个消息或事件	sendOrderPaidMessage()
receive	接受消息或时间	receiveOrderPaidMessgae()
respond	响应用户动作	responseOrderListItemClicked()
find	查找对象	findNewSupplier()
update	更新对象	updateCommission()

find 方法在业务层尽量表达业务含义，比如 findUnsettledOrders()，查询未结算订单，而不要 findOrdersByStatus()。 数据访问层，find,update 等方法可以表达要执行的 sql，比如 findByStatusAndSupplierIdOrderByName(Status.PAID, 345)

1.5 域（field）名

1.5.1 静态常量

全大写用下划线分割，如

public static find String ORDER_PAID_EVENT = “ORDER_PAID_EVENT”;

1.5.2 枚举

全大写，用下划线分割，如

```
public enum Events {
ORDER_PAID,
ORDER_CREATED
}
```

1.5.3 其他

首字母小写，骆驼法则，如：

```
public String orderName;
```

1.6 局部变量名

参数和局部变量名首字母小写，骆驼法则。尽量不要和域冲突，尽量表达这个变量在方法中的意义。

2. 代码格式

使用 tab 缩进源代码。 使用 alt+shift+f （eclipse）来格式化代码,注：格式化代码后还需手动来调下。

2.1 源文件编码

源文件使用 utf-8 编码，结尾用 unix n 分格。

2.2 行宽

行宽度不要超过 80。Eclipse 标准

2.3 包的导入

删除不用的导入，尽量不要使用整个包的导入。在 eclipse 下经常使用快捷键 ctrl+shift+o 修正导入。

2.4 类格式

2.5 域格式

每行只能声明一个域。
域的声明用空行隔开。

2.5 方法格式

2.6 代码块格式

2.6.1 缩进风格

大括号的开始在代码块开始的行尾，闭合在和代码块同一缩进的行首，同一层次的代码要保持整齐，例如：

```
Transaction transaction=null;
if ((taskFlag==null ||(taskFlag!=null && "1".equals(taskFlag
    if (launchModeNo != null && (launchModeNo.equals("2")||l
        lcIssueForm.setHandleFlag("Auto");
    } else {// 手工发起
        lcIssueForm.setHandleFlag("Hand");
    }
    String LCNo ="";
    if(launchModeNo != null && (launchModeNo.equals("3")||la
        LCNo = request.getParameter("LCNo");
    }
    if(LCNo == null || LCNo.equals("")){
        LCNo = bizNo;
    }
    mapData = this.getImportationManager()
            .getLCIssueHandleData(LCNo, transactionOrgNo,
                    txnSerialNo, lcIssueForm.getHandleFlag()
                    user, bizNo);
    lcIssueForm.setOldBizNo(bizNo);

    log.debug("mapData:"+mapData);
    if (bizNo != null && !bizNo.equals("")) {
```

2.6.2 空格的使用

2.6.2.1 表示分割时用一个空格

不能这样：

```
if      (              a >       b    )          {
    //do something here
};
```

2.6.2.2 运算符两边用一个空格隔开

例如：

```
if(launchModeNo != null && (launchModeNo.equals("3")||lau
    LCNo = request.getParameter("LCNo");
}
if(LCNo == null || LCNo.equals("")){
    LCNo = bizNo;
}
```

不要写成：

```
LCCorpInfo lcCorpInfo = (LCCorpInfo) mapData.get("LCCorpInfo");
if(lcCorpInfo!=null){
    lcIssueForm.setLcCorpInfo(lcCorpInfo);
    if(lcCorpInfo.getGoodsType()!=null){
        ShopType shopType = this.getParameterManager().getShopType(lc
        if(shopType!=null){
            lcIssueForm.setGoodsType(shopType.getShopTypeName());
        }else{
            log.error("没有找到对应的商品大类，lcCorpInfo.getGoodsType()="+lcC
        }

    }
}
```

2.6.2.3 逗号语句后如不换行，紧跟一个空格

如下：

```
call(a, b, c);
```

不能如下：

```
call(a,b,c);
```

2.6.3 空行的使用

空行可以表达代码在语义上的分割，注释的作用范围，等等。将类似操作，或一组操作放在一起不用空行隔开，而用空行隔开不同组的代码， 如图：

```
public String getDateTime(Date date,String format) {
    SimpleDateFormat sdf = new SimpleDateFormat(format);
    return sdf.format(date);
}
/**
 * 文件编号
 * @return
 * @throws Exception
 */
public String getFileNO() throws Exception{
    return new SerialNoManager().getSerialNo(cfg,"finance_timing_count");
}
/**
 * 处理返回文件
 *
 * @param path 文件路径名
 * @param content 写入内容
 *
 */
```

上例中的空行，使注释的作用域很明显.

- 连续两行的空行代表更大的语义分割。
- 方法之间用空行分割（尽量用一行空行）

- 域之间用空行分割
- 超过十行的代码如果还不用空行分割，就会增加阅读困难

3. 注释规范

3.1 注释 vs 代码

- 注释宜少二精，不宜多而滥，更不能误导
- 命名达意，结构清晰， 类和方法等责任明确，往往不需要，或者只需要很少注释，就可以让人读懂；相反，代码混乱，再多的注释都不能弥补。所以，应当先在代码本身下功夫。
- 不能正确表达代码意义的注释，只会损害代码的可读性。
- 过于详细的注释，对显而易见的代码添加的注释，罗嗦的注释，还不如不写。
- 注释要和代码同步，过多的注释会成为开发的负担

3.2 Java Doc

表明类、域和方法等的意义和用法等的注释，要以 javadoc 的方式来写。Java Doc 是个类的使用者来看的，主要介绍 是什么，怎么用等信息。凡是类的使用者需要知道，都要用 Java Doc 来写。非 Java Doc 的注释，往往是个代码的维护者看的，着重告述读者为什么这样写，如何修改，注意什么问题等。 如下：类

```
/**
 * 为XXX写的测试类
 * 2014-12-03
 * @author changxiaoyong
 *
 */
public class Test {

}
```

3.3 块级别注释

3.3.1 块级别注释，单行时用 //，多行时用 /* .. */。

3.3.2 较短的代码块用空行表示注释作用域

3.3.3 较长的代码块要用

```
/*------ start: ------*/
```

和

```
/*-------- end: -------*/
```

包围

3.4 行内注释

行内注释用 // 写在行尾

4 最佳实践和禁忌

4.1 每次保存的时候，都让你的代码是最美的

程序员都是懒惰的，不要想着等我完成了功能，再来优化代码的格式和结构，等真的把功能完成，很少有人会再愿意回头调整代码。

4.2 使用 log 而不是 System.out.println()

log 可以设定级别，可以控制输出到哪里，容易区分是在代码的什么地方打印的，而 System.out.print 则不行。而且，System.out.print 的速度很慢。所以，除非是有意的，否则，都要用 log。至少在提交到 svn 之前把 System.out.print 换成 log。

4.3 每个 if while for 等语句，都不要省略大括号{}

看下面的代码：

```
if (a > b)
    a++;
```

如果在以后维护的时候，需要在 a > b 时，把 b++，一步小心就会写成：

```
if (a > b)
    a++;
    b++;
```

这样就错了，因为无论 a 和 b 是什么关系，b++都会执行。 如果一开始就这样写：

```
if (a > b)  {
    a++;
}
```

相信没有哪个笨蛋会把 b++添加错的。而且，这个大括号使作用范围更明显，尤其是后面那行很长要折行时。

4.4 善用 TODO:

在代码中加入 //TODO: ，大部分的 ide 都会帮你提示，让你知道你还有什么事没有做。比如：

```
if (order.isPaid()) {
    //TODO: 更新订单
}
```

4.5 在需要留空的地方放一个空语句或注释，告述读者，你是故意的

比如：

```
if (!exists(order)) {
    ;
}
```

或：

```
if (!exists(order)) {
    //nothing to do
}
```

4.6 不要再对 boolean 值做 true false 判断

比如：

```
if (order.isPaid() == true) {
    // Do something here
}
```

不如写成：

```
if (order.isPaid()) {
    //Do something here
}
```

后者读起来就很是 if order is paid, …. 要比 if order’s isPaid method returns true, … 更容易理解

4.7 减少代码嵌套层次

代码嵌套层次达 3 层以上时，一般人理解起来都会困难。下面的代码是一个简单的例子：

```
public void demo(int a, int b, int c) {
    if (a > b) {
        if (b > c) {
            doJobA();
        } else if (b < c) {
            doJobB()
```

```
            }
        } else {
            if (b > c) {
                if (a < c) {
                    doJobC();
                }
            }
        }
    }
```

减少嵌套的方法有很多：

- 合并条件
- 利用 return 以省略后面的 else
- 利用子方法

比如上例，合并条件后成为：

```
public void demo(int a, int b, int c) {
    if (a > b && b > c) {
        doJobA();
    }
    if (a > b && c > b) {
        doJobB();
    }
    if (a <= b && c < b && a < c) {
        doJobC();
    }
}
```

如果利用 return 则成为：

```
public void demo(int a, int b, int c) {
    if (a > b) {
        if (b > c) {
            doJobA();
            return;
        }
        doJobB()
        return;
```

```
    }

    if (b > c) {
        if (a < c) {
            doJobC();
        }
    }
}
```

利用子方法，就是将嵌套的程序提取出来放到另外的方法里。

4.8 程序职责单一

关注点分离是软件开发的真理。人类自所以能够完成复杂的工作，就是因为人类能够将工作分解到较小级别的任务上，在做每个任务时关注更少的东西。让程序单元的职责单一，可以使你在编写这段程序时关注更少的东西，从而降低难度，减少出错。

4.9 变量的声明，初始化和被使用尽量放到一起

比方说如下代码：

```
int orderNum= getOrderNum();
//do something withou orderNum here
call(orderNum);
```

上例中的注释处代表了一段和 orderNum 不相关的代码。orderNum 的声明和初始化离被使用的地方相隔了很多行的代码，这样做不好，不如这样：

```
//do something withou orderNum here
int orderNum= getOrderNum();
call(orderNum);
```

4.10 缩小变量的作用域

能用局部变量的，不要使用实例变量，能用实例变量的，不要使用类变量。变量的生存期越短，以为着它被误用的机会越小，同一时刻程序员要关注的变量的状态越少。实例变量和类变量默认都不是线程安全的，局部变量是线程安全的。比如如下代码：

```
public class OrderPayAction{
    private Order order;

    public void doAction() {
        order = orderDao.findOrder();
```

```
        doJob1();
        doJob2();
    }

    private void doJob1() {
        doSomething(order);
    }

    private void doJob2() {
        doOtherThing(order);
    }
}
```

上例中 order 只不过担当了在方法间传递参数之用，用下面的方法更好：

```
public class OrderPayAction{
    public void doAction() {
        order = orderDao.findOrder();
        doJob1(order);
        doJob2(order);
    }

    private void doJob1(Order order) {
        doSomething(order);
    }

    private void doJob2(Order order) {
        doOtherThing(order);
    }
}
```

4.11 尽量不要用参数来带回方法运算结果

比如：

```
public void calculate(Order order) {
    int result = 0;
    //do lots of computing and store it in the result
    order.setResult(result);
}
```

```
public void action() {
    order = orderDao.findOrder();
    calculate(order);
    // do lots of things about order
}
```

例子中 calculate 方法通过传入的 order 对象来存储结果， 不如如下写：

```
public int calculate(Order order) {
    int result = 0;
    //do lots of computing and store it in the result
    return result;
}


public void action() {
    order = orderDao.findOrder();
    order.setResult(calculate(order));
    // do lots of things about order
}
```

Python 代码风格

代码布局

缩进

每级缩进用4个空格

连续行的折叠元素应该对齐

```
# 与起始定界符对齐：
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 使用更多的缩进，以区别于其他代码
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂时，应该增加一级缩进
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)

# 悬挂不一定要4个空格
foo = long_function_name(
```

```
  var_one, var_two,
  var_three, var_four)
```

if语句条件块太长需要写成多行.

值得注意的是两个字符组成的关键字（例如if），加上一个空格，加上开括号，为后面的多行条件创建了一个4个空格的缩进。这会给嵌入if内的缩进语句产生视觉冲突，因为它们也被缩进4个空格。

这个PEP没有明确如何（是否）进一步区分条件行和if语句内的行。这种情况下，可以接受的选项包括，但不仅限于：

```
# 不增加额外的缩进
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 添加一行注释，这将为支持语法高亮的编辑器提供一些区分
if (this_is_one_thing and
    that_is_another_thing):
    # 当两个条件都是真，我们将要执行
    do_something()

# 在换行的条件语句前，增加额外的缩进
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

多行结构中的结束花括号/中括号/圆括号应该是最后一行的第一个非空白字符

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

或者是最后一行的第一个字符

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

制表符还是空格

空格是最优先的缩进方式

当已经使用制表符是，应该保持一致性，继续使用制表符
Python 3不允许混合使用制表符和空格来缩进。
Python 2的代码中混合使用制表符和空格的缩进，应该转化为完全使用空格。
调用python命令行解释器时使用-t选项,可对代码中不合法得混合制表符和空格发出警告(warnings)。使用-tt时警告(warnings)将变成错误(errors).这些选项是被高度推荐的.

最大行长度

限制所有行最多79个字符。
下垂的长块结构限制为更少的文本（文档字符串或注释），行的长度应该限制在72个字符。
限制编辑器窗口宽度使得并排打开多个文件成为可能,并且使用代码审查工具显示相邻列的两个版本工作正常。
绝大多数工具的默认折叠会破坏代码的可视化结构，使其更难以理解。编辑器中的窗口宽度设置为80个字符。即使该工具将在最后一列中标记字形。一些基于网络的工具可能不会提供动态的自动换行。
有些团队强烈喜欢较长的行长度。对于代码维护完全或主要由一个团队的，可以在这个问题上达成协议，象征性的将行长度从80个字符增加到100个字符（有效地增加最大长度到99个字符）也是可以的,提供注释和文档字符串仍是72个字符。
Python标准库采取保守做法，要求行限制到79个字符（文档字符串/注释到72个字符）。
折叠长行的首选方法是使用Pyhon支持的圆括号,方括号(brackets)和花括号(braces)内的行延续。长行可以在表达式外面使用小括号来变成多行。连续行使用反斜杠更好。
反斜杠有时可能仍然是合适的。例如，长的多行的with语句不能用隐式续行，可以用反斜杠：

```
# 反斜杠有时可能仍然是合适的。例如，长的多行的with语句不能用隐式续行，可以用反斜杠：
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

另一种类似的情况是在assert语句中
确认恰当地缩进了延续的行

换行应该在二元操作符前面还是后面

几十年来推荐的风格是在二元操作符后换行。但这会通过两种方式影响可读性:操作符往往分散在屏幕的不同的列上,并且每个操作符都被移动到远离其操作数。在这里,眼睛要做额外的工作看清哪些数是添加或减去:

```
# 坏的: 操作符远离它们的操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为了解决这个可读性问题,数学家和出版商遵循相反的约定。

Donald Knuth在他的《电脑和排版》系列中解释了传统规则:“尽管后一段总是打破内公式二进制操作和关系,显示公式总是在二进制操作前换行”。
以下的数学运算传统通常可以使得结果更加具有可读性

```
# 好的: 易于匹配操作数和操作符
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
#二元运算符首选的换行位置在操作符后面
class Rectangle(Blob):
    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
                color == 'red' and emphasis == 'strong' or \
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

在Python代码中,允许打破之前或之后一个二元运算符的规则,只要与当前惯例是一致的。为新代码Knuth的风格。

空行

用两行空行分割顶层函数和类的定义.
类内方法的定义用单个空行分割.
额外的空行可被用于(保守的(sparingly))分割一组相关函数(groups of related functions).
在一组相关的单句中间可以省略空行.(例如.一组哑元(a set of dummy implementations)).
在函数中使用空行时,请谨慎的用于表示一个逻辑段落(indicate logical sections).
Python接受contol-L(即^L)换页符作为空格;Emacs(和一些打印工具) 视这个字符为页面分割符,因此在你的文件中,可以用他们来为相关片段(sections)分页。
注意，一些编辑器和基于Web的代码查看器可能不能识别control-L是换页，将显示另外的字形。

源文件编码

Python核心发布中的代码应该始终使用UTF-8（或Python2中用ASCII）。
文件使用ASCII（Python2中）或UTF-8（Python3中）不应有编码声明。
在标准库中，非默认编码仅用于测试目的或注释或文档字符串需要提及包含非ASCII字符的作者名；否则，使用\x，\u，\U，或\N是字符串中包含非ASCII数据的首先方式。

Python3.0及以上版本，为标准库（参见PEP 3131）规定以下策略：Python标准库中的所有标识符必须使用ASCII标识符，在可行的地方使用英文单词（在很多例子中，使用非英文的缩写和专业术语）。
另外，字符串和注释必须用ASCII。仅有的例外是（a）测试非ASCII的特点，（b）测试作者名。不是基于拉丁字母表的作者名必须提供一个他们名字的拉丁字母表的音译。
开源项目面向全球，鼓励采用统一策略。

导入

```
# 通常应该在单独的行中导入(Imports)
import os
import sys

# 这样也是可以的
from subprocess import Popen, PIPE
```

Imports 通常被放置在文件的顶部,仅在模块注释和文档字符串之后,在模块的全局变量和常量之前.
Imports 应该有顺序地成组安放.

- 标准库的导入(Imports)
- 相关的第三方导入(即,所有的email包在随后导入)
- 特定的本地应用/库导入(imports)
- 你应该在每组导入之间放置一个空行.

把任何相关all规范放在导入之后。
推荐绝对导入，因为它们更易读，并且如果导入系统配置的不正确（例如当包中的一个目录在sys.path的最后）它们有更好的表现（或者 至少给出更好的错误信息）：

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

明确的相对导入可以被接受用来替代绝对导入，特别是处理复杂的包布局时，绝对导入过于冗长。

```
from . import sibling
from .sibling import example
```

标准库代码应该避免复杂包布局并使用绝对导入。
隐式的相对导入应该永远不被使用，并且在Python3中已经移除。
从一个包含类的模块中导入类时，下面这样通常是好的写法：

```
from myclass import MyClass
from foo.bar.yourclass import YourClass

# 如果这种写法导致本地名字冲突，那么就这样写：

import myclass
import foo.bar.yourclass
#并使用“myclass.MyClass”和“foo.bar.yourclass.YourClass”来访问。
```

避免使用通配符导入（from <模块名> import *），因为这样就不清楚哪些名字出现在命名空间，这会让读者和许多自动化工具困惑。
通配符导入有一种合理的使用情况，重新发布一个内部接口作为一个公共API的一部分（例如，重写一个纯Python实现的接口，该接口定义从一个可选的加速器模块并且哪些定义将被重写提前并不知道）。
用这种方式重新命名，下面的有关公共和内部接口的指南仍适用。

字符串引号

Python中，单引号字符串和双引号字符串是一样的,本PEP不建议如此,建议选择一个并坚持下去。
当一个字符串包含单引号字符或双引号字符时,使用另一种字符串引号来避免字符串中使用反斜杠。这可以提高可读性。
三引号字符串，总是使用双引号字符，与PEP 257 文档字符串规范保持一致。

表达式和语句中的空格

Guido不喜欢在以下地方出现空格

以下情况避免使用多余的空格

```
# 紧挨着圆括号,方括号和花括号的
spam(ham[1], {eggs: 2})

# 紧贴在逗号,分号或冒号前的
if x == 4: print x, y; x, y = y, x


# 在切片中冒号像一个二元操作符，冒号两侧应该有相同数量的空格（把它看作最低优先级的操作符）。
# 在一个扩展切片中，两个冒号必须有相等数量的空格。
# 例外：当一个切片参数被省略时，该空格也要被省略。


ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]

# 紧贴着函数调用的参数列表前开式括号(open parenthesis )的
spam(1)

# 紧贴在索引或切片(slicing?下标?)开始的开式括号前的
dct['key'] = lst[index]

# 在赋值(或其它)运算符周围，不要为了与其他的赋值（或其它）操作符对齐，使用不止一个空格。
x = 1
y = 2
```

```
long_variable = 3
```

其它建议

避免尾随空格。

因为它通常是无形的,它可能会让人很困惑:如一个反斜杠,后跟一个空格和一个换行符不算作一行延续标记。有些编辑器不保护它，并且许多项目(如CPython本身)导向钩子,拒绝它。

始终在这些二元运算符两边放置一个空格:赋值(=)，比较(==，<，>，!=，<>，<=,>=，in，not in，is，is not)，布尔运算 (and，or，not).

如果使用了不同优先级的操作符，在低优先级操作符周围增加空格（一个或多个）。不要使用多于一个空格，二元运算符两侧空格数量相等。

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# 不要在用于指定关键字参数或默认参数值的'='号周围使用空格,例如:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

# 函数注释应该对冒号使用正常规则，并且，如果存在 -> ，两边应该有空格
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...

# 结合一个参数注释和默认值时,在 = 符号两边使用空格(仅仅当那些参数同时有注释和一个默认值时)。
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...

# 不要将多条语句写在同一行上.
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()

# 尽管有时可以将if/for/while和一小段主体代码放在一行,但在一个有多条子句的语句中不要如此。避免折叠长行!
# 最好不要
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

注释

同代码不一致的注释比没注释更差.当代码修改时,始终优先更新注释!

注释应该是完整的句子. 如果注释是一个短语或句子,首字母应该大写, 除非他是一个以小写字母开头的标识符(永远不要修改标识符的大小写).
如果注释很短,最好省略末尾的句号。注释块通常由一个或多个由完整句子构成的段落组成,每个句子都应该以句号结尾.
你应该在句末的句号后使用两个空格,以便使Emacs的断行和填充工作协调一致(译按:应该说是使这两种功能正常工作,". "给出了文档结构的提示).
用英语书写时,断词和空格是可用的.
非英语国家的Python程序员:请用英语书写你的注释,除非你120%的确信 这些代码不会被不懂你的语言的人阅读.
"""
"""

我就是坚持全部使用中文来注释，真正要发布脚本工具时，再想E文的;
开发时每一瞬间都要用在思量中，坚决不用在E文语法，单词的回忆中!
-- ZoomQUiet
约定使用统一的文档化注释格式有利于良好习惯和团队建议!文档化开发注释规范

注释块

注释块通常应用于跟随着一些(或者全部)代码并和这些代码有着相同的缩进层次. 注释块中每行以'#'和一个空格开始(除非他是注释内的缩进文本).
注释块内的段落以仅含单个'#'的行分割.

行内注释

行内注释应该谨慎使用.
一个行内注释是和语句在同一行的注释.行内注释应该至少用两个空格和语句分开. 它们应该以'#'和单个空格开始.
如果行内注释指出的是显而易见的，那么它就是不必要的。

```
# 不要这样做:
x = x + 1                 # 增加 x

# 但有时，这样是有用的:
x = x + 1                 # 补偿border
```

文档字符串

应该一直遵守在PEP 257中编写好的文档字符串(又名"docstrings")的约定,
"""
"""

Documentation Strings-- 文档化字符 ;
为配合 pydoc;epydoc,Doxygen等等文档化工具的使用,类似于MoinMoin 语法,约定一些字符,
以便自动提取转化为有意义的文档章节等等文章元素!
-- Zoomq
"""

为所有公共模块,函数,类和方法编写文档字符串.文档字符串对非公开的方法不是必要的,但你应该有一个描述这个方法做什么的注释.这个注释应该在"def"这行后.

PEP 257 描述了好的文档字符串的约定.一定注意,多行文档字符串结尾的 \"\"\" 应该单独成行,例如:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

对单行的文档字符串,请保持结尾的"""在同一行.

版本注记

如果你要将RCS或CVS的杂项(crud)包含在你的源文件中,按如下做.

```
__version__ = "$Revision$"
# $Source: E:/cvsroot/python_doc/pep8.txt,v $
```

这个行应该包含在模块的文档字符串之后,所有代码之前,上下用一个空行分割.

对于CVS的服务器工作标记更应该在代码段中明确出它的使用

如：在文档的最开始的版权声明后应加入如下版本标记：

```
# 文件：$id$
# 版本： $Revision$
```

这样的标记在提交给配置管理服务器后，会自动适配成为相应的字符串，如：

```
# 文件：$Id: ussp.py,v 1.22 2004/07/21 04:47:41 hd Exp $
# 版本： $Revision: 1.4 $
```

----HD

命名约定

Python库的命名约定有点混乱,所以我们将永远不能使之变得完全一致--- 不过还是有公认的命名规范的.

新的模块和包(包括第三方的框架)必须符合这些标准,但对已有的库存在不同风格的, 保持内部的一致性是首选的.

根本原则

用户可见的API的公开部分的名称，应该体现用法而不是实现。

描述:命名风格

有许多不同的命名风格.以下的有助于辨认正在使用的命名风格,独立于它们的作用.

以下的命名风格是众所周知的:

- b (单个小写字母)
- B (单个大写字母)
- 小写串 如:getname
- 带下划的小写串 如:_getname,lower_case_with_underscores
- 大写串 如:GETNAME
- 带下划的大写串 如:_GETNAME,UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords(首字母大写单词串) (或 CapWords, CamelCase/驼峰命名法 -- 这样命名是由于它的字母错落有致的样子而来的.这有时也被当作StudlyCaps. 如:GetName
  - 注意：当CapWords中使用缩写，大写所有的缩写字母。因此HTTPServerError优于HttpServerError。
- mixedCase (混合大小写串)(与首字母大写串不同之处在于第一个字符是小写 如:getName)

• Capitalized_Words_With_Underscores(带下划线的首字母大写串) (丑陋!)
还有一种使用特别前缀的风格,用于将相关的名字分成组。Python中很少这样用,但是出于完整性要提一下.
例如,os.stat()函数返回一个元祖,它的元素名字通常类似st_mode,st_size,st_mtime等等这样。(这样做是为了强调与POSIX系统调用结构体一致,这有助于程序员熟悉这些。)
X11库的所有的公开函数以X开头。Python中,这种风格通常认为是不必要的,因为属性名和函数名以对象名作前缀,而函数名以模块名作前缀。
另外,以下用下划线作前导或结尾的特殊形式是被公认的(一般可以与任何约定相结合):
_single_leading_underscore(以一个下划线作前导): 弱的"内部使用(internal use)"标志.
(例如,"from M import *"不会导入以下划线开头的对象).
single_trailing_underscore_(以一个下划线结尾): 用于避免与Python关键词的冲突,例如.
"Tkinter.Toplevel(master, class_='ClassName')".
__double_leading_underscore(双下划线): 从Python 1.4起为类私有名.调用时名称改变(类FooBar中, __boo变成 _FooBar__boo; 见下文)。
__double_leading_and_trailing_underscore__: 特殊的(magic) 对象或属性,存在于用户控制的(user-controlled)名字空间.
例如:__init__, __import__ 或 __file__.
有时它们被用户定义, 用于触发某个特殊行为(magic behavior)(例如:运算符重载);
有时被构造器(infrastructure)插入,以便自己使用或为了调试. 因此,在未来的版本中,构造器(松散得定义为Python解释器和标准库) 可能打算建立自己的魔法属性列表,用户代码通常应该限制将这种约定作为己用. 欲成为构造器的一部分的用户代码可以在下滑线中结合使用短前缀,例如.
__bobo_magic_attr__.

说明:命名约定

a. 应避免的名字
永远不要用字符'l'(小写字母el(就是读音,下同)),'O'(大写字母oh),或'I'(大写字母eye)作为单字符的变量名.
在某些字体中,这些字符不能与数字1和0分开.当想要使用'l'时, 用'L'代替它.

b. 包和模块名
模块应该是简短的,全小写的名字. 可以在模块名中使用下划线来提高可读性.
Python包名也应该是简短的,全小写的名字, 不鼓励使用下划线。
当一个用C或C++写的扩展模块有一个伴随的Python模块,这个Python模块提供了一个更高层(例如, 更面向对象)的接口时,C/C++模块有一个前导下划线(如:_socket)

c. 类名
几乎没有例外, 类名总是使用首字母大写单词串(CapWords)的约定.

d. 异常名
因为异常应该是一个类, 类的规范也适用这里。无论怎样, 你应该在异常名中使用后缀"Error"(如果实际是一个错误).

似乎内建(扩展)的模块使用"error"(例如:os.error), 而Python模块通常用"Error" (例如: xdrlib.Error).
e. 全局变量名
(让我们希望这些变量打算只被用于模块内部) 这些约定与那些用于函数的约定差不多.
被设计可以通过"from M import *"来使用的那些模块, 应该使用all机制防止导出全局变量，或使用加前缀的旧规则，在那些不想被导入的全局变量(还有内部函数和类)前加一个下划线.
f. 函数名
函数名应该为小写,可以将单词用下划线分开以增加可读性.
mixedCase混合大小写 仅被允许用于这种风格已经占优势的上下文(如: threading.py) 以便保持向后兼容.
g. 函数和方法参数
总是使用self做实例方法的第一个参数。
总是使用cls做类方法的第一个参数。
如果一个函数的参数名与保留关键字冲突，最好是为参数名添加一个后置下划线而不是使用缩写或错误的拼写。因此class_ 比clss好。（也许使用同义词来避免更好。）。
h. 方法名和实例变量
大体上和函数相同:使用小写单词,必要时用下划线分隔增加可读性.
仅为不打算作为类的公共接口的内部方法和实例变量使用一个前导下划线.
为了避免和子类命名冲突，使用两个前导下划线调用Python的名称改编规则。
Python用类名改编这些名字：如果类Foo有一个属性名为a，通过Foo.a不能访问。（执著的用户可以通过调用Foo._Foo__a来访问。）通常，两个前导下划线仅用来避免与子类的属性名冲突。
注意：关于__names的使用存在一些争论（见下文）。
i. 常量
常量通常在模块级别定义，并且所有的字母都是大写，单词用下划线分开。例如: MAX_OVERFLOW 和 TOTAL。
j. 继承的设计
确定类的方法和实例变量（统称为：“属性”）是否公开。如果有疑问，选择非公开；之后把其变成公开比把一个公开属性改成非公开要容易。
公开属性是那些你期望与你的类不相关的客户使用的，根据你的承诺来避免向后不兼容的变更。非公开属性是那些不打算被第三方使用的；你不保证非公开属性不会改变甚至被删除。
此处没有使用术语“private”，因为Python中没有真正私有的属性（没有通常的不必要的工作）。
属性的另一个类别是“API子集”的一部分（在其它语言常被称为“protected”）。某些类被设计为基类，要么扩展，要么修改某些方面的类的行为。在设计这样的类的时候，要注意明确哪些属性是公开的，哪些是API子类的一部分，哪些是真正只在你的基类中使用。
清楚这些之后，以下是 Pythonic 的指南：
公开属性没有前导下划线。

如果公开属性名和保留关键字冲突，给属性名添加一个后置下划线。这比缩写或拼写错误更可取。（然而，尽管有这样的规定，对于任何类的变量或参数，特别是类方法的第一个参数，‘cls’是首选的拼写方式）
注1：参见上面对类方法的参数名的建议。
对于简单的公开数据属性，最好只暴露属性名，没有复杂的访问器/修改器方法。记住，Python为未来增强提供了一条简单的途径，你应该发现简单的数据属性需要增加功能行为。在这种情况下，使用属性来隐藏简单数据属性访问语法后面的功能实现。
注1：特性仅工作于新风格的类。
注2：尽量保持功能行为无副作用，尽管副作用如缓存通常是好的。
注3：计算开销较大的操作避免使用特性，属性注解使调用者相信访问（相对）是廉价的。
如果确定你的类会被子类化，并有不想子类使用的属性，考虑用两个前导下划线无后置下划线来命名它们。这将调用Python的名称改编算法，类名将被改编为属性名。当子类不无意间包括相同的属性名时，这有助于帮助避免属性名冲突。
注1：注意改编名称仅用于简单类名，如果一个子类使用相同的类名和属性名，仍然会有名字冲突。
注2：名称改编会带来一定的不便，如调试和getattr()。然而，名称改编算法有良好的文档，也容易手工执行。
注3：不是每个人都喜欢名称改编。尝试平衡避免意外的名称冲突和高级调用者的可能。

公共和内部接口

任何向后兼容性保证只适用于公共接口。因此，重要的是用户能够清楚地区分公开和内部接口。
文档接口被认为是公开的，除非文档明确声明他们是临时或内部接口，免除通常的向后兼容保证。所有非文档化的接口应假定为内部接口。
为了更好的支持自省，模块应该使用all属性显示声明他们公开API的名字， all设置为一个空列表表示该模块没有公开API。
即使all设置的适当，内部接口（包，模块，类，函数，属性或者其它名字）仍应以一个前导下划线作前缀。
一个接口被认为是内部接口，如果它包含任何命名空间（包，模块，或类）被认为是内部的。
导入名被认为是实现细节。其它模块必须不依赖间接访问这个导入名，除非他们是一个明确的记录包含模块的API的一部分，例如os.path或包的init模块，从子模块暴露功能。

程序设计建议

- 编写的代码应该不损害其他方式的Python实现（PyPy，Jython，IronPython，Cython，Psyco等等）。

例如，不要依赖CPython的高效实现字符串连接的语句形式 += b或a = a + b。这种优化即使在CPython里也是脆弱的（它只适用于某些类型），并且在不使用引用计数的实现中它完全不存在。在库的性能易受影响的部分，应使用’’.join()形式。这将确保跨越不同实现的连接发生在线性时间。

- 与单值，比如None比较，使用is或is not，不要用等号操作符。

同样，如果你真正的意思是if x is not None，谨防编写if x。例如，当测试一个默认是None的变量或参数是否被置成其它的值时。这个其它值可能是在布尔上下文为假的类型（例如容器）。

• 使用 is not 操作符而不是 not...is。虽然这两个表达式的功能相同，前者更具有可读性并且更优。

好:
```
if foo is not None:
```
坏:
```
if not foo is None:
```

• 当实现有丰富的比较的排序操作时，最好实现所有六个操作符（__eq__，__ne__，__lt__，__le__，__gt__，__ge__）而不是依靠其它代码只能进行一个特定的比较。

为了减少所涉及的工作量，functools.total_ordering()装饰器提供了一个工具来生成缺失的比较函数。

PEP 207表明，Python假定自反性规则。因此，编译器可以交换y > x和x < y，y >= x和x <= y，也可以交换参数x == y和x != y。sort()和min()操作保证使用< 操作符并且max()功能使用> 操作符。不管怎样，最好实现所有六个操作，这样在其它上下文就不会产生混淆了。

• 总是使用def语句而不是使用赋值语句绑定lambda表达式到标识符上。

风格良好：
```
def f(x): return 2*x
```
风格不良：
```
f = lambda x: 2*x
```

第一种形式意味着所得的函数对象的名称是‘f’而不是一般的‘<lambda>’。这在回溯和字符串表示中更有用。

赋值语句的使用消除了lambda表达式可以提供显示def声明的唯一好处（例如它可以嵌在更大的表达式里面）。

• 从Exception而不是BaseException中派生出异常。直接继承BaseException是保留那些捕捉几乎总是错的异常的。

设计异常层次结构基于区别，代码可能需要捕获异常，而不是捕获产生异常的位置。

旨在以编程方式回答问题“出了什么问题？”，而不是只说“问题产生了”（参见PEP 3151对这节课学习内置异常层次结构的一个例子）

类的命名规则适用于此，只是当异常确实是错误的时候，需要在异常类名添加“Error”后缀。用于非本地的流控制或其他形式的信号的非错误的异常，不需要特殊的后缀。

• 适当使用异常链。Python3中，“raise X from Y”用来表明明确的更换而不失去原来追踪到的信息。

当故意替换一个内部异常（Python 2中使用“raise X”而Python 3.3+中使用“raise X from None”），

确保相关的细节被转移到新的异常中（比如当转换KeyError为AttributeError时保留属性名，或在新的异常消息中嵌入原始异常的文本）。

• Python 2中产生异常，使用raise ValueError('message')替换老的形式raise ValueError,'message'。

后一种形式是不合法的Python 3语法。
目前使用的形式意味着当异常的参数很长或包含格式化字符传时，多亏了小括号，不必再使用续行符。
捕获异常时，尽可能提及特定的异常，而不是使用空的except：子句。
例如，使用：

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

• 空的except：子句将捕获SystemExit和KeyboardInterrupt异常，这使得很难用Control-C来中断程序，也会掩饰其它的问题。

如果想捕获会导致程序错误的所有异常，使用except Exception:（空异常相当于except BaseException:）
一条好的经验法则是限制使用空‘except’子句的两种情况：
1 如果异常处理程序将打印或记录跟踪；至少用户将会意识到有错误发生。
2 如果代码需要做一些清理工作，但是随后让异常用raise抛出。处理这种情况用try...finally更好。
当用一个名字绑定捕获异常时，更喜欢Python2.6中添加的明确的名称绑定语法。

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

这是Python3中唯一支持的语法，并避免与旧的基于逗号的语法有关的歧义问题。
捕获操作系统异常时，优先使用 Python 3.3引进的明确的异常层次，通过errno值自省。
另外，对于所有的try/except子句，限制try子句内只有绝对最少代码量。这避免掩盖错误。
风格良好：

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

风格不良：

```
try:
    # 很多代码
    return handle_value(collection[key])
except KeyError:
    # 捕获由handle_value()抛出的KeyError
    return key_not_found(key)
```

当一个资源是一个局部特定的代码段，它使用后用with语句确保迅速可靠的将它清理掉。也可以使用try/finally语句。

无论何时做了除了获取或释放资源的一些操作，都应该通过单独的函数或方法调用上下文管理器。例如：

风格良好：

```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

风格不良：

```
with conn:
    do_stuff_in_transaction(conn)
```

后者的例子没有提供任何信息表明enter和__exit__方法做了什么，除了事务结束后关闭连接。 在这种情况下，明确是很重要的。

返回语句保持一致。函数中的所有返回语句都有返回值，或都没有返回值。

如果任意一个返回语句有返回值，那么任意没有返回值的返回语句应该明确指出return None，并且一个显式的返回语句应该放在函数结尾（如果可以）。

风格良好：

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

风格不良：

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)
```

使用字符串方法代替string模块。

字符串方法总是更快并且与unicode字符串使用相同的API。如果必须向后兼容Python2.0以前的版本，无视这个原则。

使用 ''.startswith() 和 ''.endswith() 代替字符串切片来检查前缀和后缀。startswith()和endswith()更清晰，并且减少错误率。例如：

风格良好：`if foo.startswith('bar'):`

风格不良：`if foo[:3] == 'bar':`

对象类型的比较使用isinstance()代替直接比较类型。

风格良好：`if isinstance(obj, int):`

风格不良：`if type(obj) is type(1):`

当检查一个对象是否是字符串时，牢记它也可能是一个unicode字符串！Python 2中，str和unicode有共同的基类，basestring，所以你可以这么做：

```
if isinstance(obj, basestring):
```

注意，Python3中， unicode 和 basestring 不再存在（只有str），并且字节对象不再是string的一种（而是一个integers序列）

对于序列，（字符串，列表，元组），利用空序列是false的事实。

风格良好：

```
if not seq:
if seq:
```

风格不良：

```
if len(seq)
if not len(seq)
```

不要书写依赖后置空格的字符串。这些后置空格在视觉上无法区分，并且有些编辑器（或最近，reindent.py）将去掉他们。

不要用==来将布尔值与True或False进行比较。

风格良好：

```
if greeting:
```

风格不良：

```
if greeting == True:
```

糟糕的：

```
if greeting is True:
```

函数注释

为了包含 PEP 484 ，风格指南的函数注释部分进行了更新

为了向前兼容,Python 3中的函数注释代码应该更好地使用PEP 484语法。(在前一节中有一些注释格式的建议)。

不再鼓励之前在这个PEP中推荐的尝试性的注释风格。

然而,除了标准库,现在鼓励使用PEP 484中的尝试性的规则。例如,使用PEP 484风格类型的注释标记一个大型第三方库或应用,检查添加这些注释有多容易,并观察他们的是否会使代码更加易懂。

Python标准库应该对使用这些注释采取保守的态度,但允许在新的代码或大型重构中使用这些注释规则。

对于代码，如果想要用函数注释的不同用法，推荐使用如下格式的注释:

```
# type: ignore
```

在靠近文件的顶部,这告诉类型检查器忽略所有注释。(更详细的关闭类型检查器报错的方法，可以在 PEP 484 中查找)。

类型检查是可选的,单独的工具。Python解释器在默认情况下不应该发布任何类型检查的消息，并且应该不改变他们基于注释的行为。

不想使用类型检查的用户可以自由地忽略它们。然而,可能用户的第三方库包可能想要在这些包种运行类型检查。为此，PEP 484建议使用存根文件:'.pyi' 文件被类型检查器读取相应的'.py'文件的偏好。存根文件可以通过资源和库一起发布，或在获得库作者的许可时单独发布。

HTML/CSS/JS 代码风格

样式规则

协议

嵌入式资源书写省略协议头

省略图像、媒体文件、样式表和脚本等 URL 协议头部声明
(http: , https:)。如果不是这两个声明的 URL 则不省略。

省略协议声明，使 URL 成相对地址，防止内容混淆问题和导致小文件重复下载。

```
<!-- 不推荐 -->

<script
src="http://www.google.com/js/gweb/analytics/autotrack.js"></script>

<!-- 推荐 -->

<script
src="//www.google.com/js/gweb/analytics/autotrack.js"></script>

/* 不推荐 */

.example {

  background: url(http://www.google.com/images/example);

}

/* 推荐 */

.example {

  background: url(//www.google.com/images/example);

}
```

排版规则

缩进

每次缩进两个空格。

不要用 TAB 键或多个空格来进行缩进。

```
<ul>
```

```
  <li>Fantastic

  <li>Great

</ul>

.example {

  color: blue;

}
```

大小写

只用小写字母。

所有的代码都用小写字母:适用于元素名,属性,属性值(除了文本和 CDATA),选择器，特性，特性值（除了字符串）。

```
<!-- 不推荐 -->

<A HREF="/">Home</A>

<!-- 推荐 -->

<img src="google.png" alt="Google">
```

行尾空格

删除行尾白空格。

行尾空格没必要存在。

```
<!-- 不推荐 -->

<p>What?_

<!-- 推荐 -->

<p>Yes please.
```

元数据规则

编码

用不带 BOM 头的 UTF-8 编码。

让你的编辑器用没有字节顺序标记的 UTF-8 编码格式进行编写。

在 HTML 模板和文件中指定编码 <meta charset="utf-8"> . 不需要制定样式表的编码，它默认为 UTF-8.

（更多有关于编码的信息和怎样指定它，请查看 Character Sets & Encodings in XHTML, HTML and CSS。）

注释

尽可能的去解释你写的代码。

用注释来解释代码：它包括什么，它的目的是什么，它能做什么，为什么使用这个解决方案，还是说只是因为偏爱如此呢？

（本规则可选，没必要每份代码都描述的很充分，它会增重 HTML 和 CSS 的代码。这取决于该项目的复杂程度。）

活动的条目

用 TODO 标记代办事项和正活动的条目

只用 TODO 来强调代办事项， 不要用其他的常见格式，例如 @@ 。

附加联系人(用户名或电子邮件列表)，用括号括起来，例如 TODO(contact) 。

可在冒号之后附加活动条目说明等，例如 TODO: 活动条目说明 。

{# TODO(cha.jn): 重新置中 #}

Test

<!-- TODO: 删除可选元素 -->


```
  <li>Apples</li>

  <li>Oranges</li>

</ul>
```

HTML 代码风格规则

文档类型

请使用 HTML5 标准。

HTML5 是目前所有 HTML 文档类型中的首选： <!DOCTYPE html> .

（推荐用 HTML 文本文档格式，即 text/html . 不要用 XHTML。 XHTML 格式，即 application/xhtml+xml ，有俩浏览器完全不支持，还比 HTML 用更多的存储空间。）

HTML 代码有效性

尽量使用有效的 HTML 代码。

编写有效的 HTML 代码，否则很难达到性能上的提升。

用类似这样的工具 W3C HTML validator 来进行测试。

HTML 代码有效性是重要的质量衡量标准，并可确保 HTML 代码可以正确使用。

```
<!-- 不推荐 -->

<title>Test</title>

<article>This is only a test.

<!-- 推荐 -->

<!DOCTYPE html>

<meta charset="utf-8">

<title>Test</title>
```

```
<article>This is only a test.</article>
```

语义

根据 HTML 各个元素的用途而去使用它们。

使用元素（有时候错称其为“标签”）要知道为什么去使用它们和是否正确。 例如，用 heading 元素构造标题， p 元素构造段落， a 元素构造锚点等。

根据 HTML 各个元素的用途而去使用是很重要的，它涉及到文档的可访问性、重用和代码效率等问题。

```
<!-- 不推荐 -->

<div onclick="goToRecommendations();">All recommendations</div>

<!-- 推荐 -->

<a href="recommendations/">All recommendations</a>
```

多媒体后备方案

为多媒体提供备选内容。

对于多媒体，如图像，视频，通过 canvas 读取的动画元素，确保提供备选方案。 对于图像使用有意义的备选文案（ alt ） 对于视频和音频使用有效的副本和文案说明。

提供备选内容是很重要的,原因:给盲人用户以一些提示性的文字,用 @alt 告诉他这图像是关于什么的，给可能没理解视频或音频的内容的用户以提示。

（图像的 alt 属性会产生冗余，如果使用图像只是为了不能立即用 CSS 而装饰的 ，就不需要用备选文案了，可以写 alt="" 。）

```
<!-- 不推荐 -->

<img src="spreadsheet.png">

<!-- 推荐 -->

<img src="spreadsheet.png" alt="电子表格截图">
```

关注点分离

将表现和行为分开。

严格保持结构 （标记），表现 （样式），和行为 （脚本）分离，并尽量让这三者之间的交互保持最低限度。

确保文档和模板只包含 HTML 结构， 把所有表现都放到样式表里，把所有行为都放到脚本里。

此外，尽量使脚本和样式表在文档与模板中有最小接触面积，即减少外链。

将表现和行为分开维护是很重要滴，因为更改 HTML 文档结构和模板会比更新样式表和脚本更花费成本。

```
<!-- 不推荐 -->

<!DOCTYPE html>

<title>HTML sucks</title>

<link rel="stylesheet" href="base.css" media="screen">

<link rel="stylesheet" href="grid.css" media="screen">

<link rel="stylesheet" href="print.css" media="print">

<h1 style="font-size: 1em;">HTML sucks</h1>

<p>I’ ve read about this on a few sites but now I’ m sure:

  <u>HTML is stupid!!1</u>

<center>I can’ t believe there’ s no way to control the styling of

  my website without doing everything all over again!</center>

<!-- 推荐 -->

<!DOCTYPE html>

<title>My first CSS-only redesign</title>
```

```
<link rel="stylesheet" href="default.css">

<h1>My first CSS-only redesign</h1>

<p>I’ve read about this on a few sites but today I’m actually

  doing it: separating concerns and avoiding anything in the HTML of

  my website that is presentational.

<p>It’s awesome!
```

实体引用

不要用实体引用。

不需要使用类似 — 、 ” 和 ☺ 等的实体引用，假定团队之间所用的文件和编辑器是同一编码（UTF-8）。

在 HTML 文档中具有特殊含义的字符（例如 < 和 &)为例外， 噢对了，还有 “不可见” 字符 （例如 no-break 空格）。

```
<!-- 不推荐 -->

欧元货币符号是 &ldquo;&eur;&rdquo;。

<!-- 推荐 -->

欧元货币符号是 “€”。
```

可选标签

省略可选标签（可选）。

出于优化文件大小和校验， 可以考虑省略可选标签，哪些是可选标签可以参考 HTML5 specification。

（这种方法可能需要更精准的规范来制定，众多的开发者对此的观点也都不同。考虑到一致性和简洁的原因，省略所有可选标记是有必要的。）

```
<!-- 不推荐 -->
```

```
<!DOCTYPE html>

<html>

  <head>

    <title>Spending money, spending bytes</title>

  </head>

  <body>

    <p>Sic.</p>

  </body>

</html>

<!-- 推荐 -->

<!DOCTYPE html>

<title>Saving money, saving bytes</title>

<p>Qed.
```

type 属性

在样式表和脚本的标签中忽略 type 属性

在样式表（除非不用 CSS）和脚本（除非不用 JavaScript）的标签中 不写 type 属性。

HTML5 默认 type 为 text/css 和 text/javascript 类型，所以没必要指定。即便是老浏览器也是支持的。

```
<!-- 不推荐 -->

<link rel="stylesheet" href="//www.google.com/css/maia.css"

  type="text/css">

<!-- 推荐 -->
```

```
<link rel="stylesheet" href="//www.google.com/css/maia.css">

<!-- 不推荐 -->

<script src="//www.google.com/js/gweb/analytics/autotrack.js"

  type="text/javascript"></script>

<!-- 推荐 -->

<script
src="//www.google.com/js/gweb/analytics/autotrack.js"></script>
```

HTML 代码格式规则

格式

每个块元素、列表元素或表格元素都独占一行，每个子元素都相对于父元素进行缩进。

独立元素的样式（as CSS allows elements to assume a different role per display property)，将块元素、列表元素或表格元素都放在新行。

另外，需要缩进块元素、列表元素或表格元素的子元素。

（如果出现了列表项左右空文本节点问题，可以试着将所有的 li 元素都放在一行。 A linter is encouraged to throw a warning instead of an error.)

```
<blockquote>

  <p><em>Space</em>, the final frontier.</p>

</blockquote>

<ul>

  <li>Moe

  <li>Larry

  <li>Curly

</ul>
```

```
<table>
  <thead>
    <tr>
      <th scope="col">Income
      <th scope="col">Taxes
  <tbody>
    <tr>
      <td>$ 5.00
      <td>$ 4.50
</table>
```

CSS 代码风格规则

CSS 代码有效性

尽量使用有效的 CSS 代码。

使用有效的 CSS 代码，除非是处理 CSS 校验器程序错误或者需要专有语法。

用类似 W3C CSS validator 这样的工具来进行有效性的测试。

使用有效的 CSS 是重要的质量衡量标准，如果发现有的 CSS 代码没有任何效果的可以删除，确保 CSS 用法适当。

ID 和 class 的命名

为 ID 和 class 取通用且有意义的名字。

应该从 ID 和 class 的名字上就能看出这元素是干嘛用的，而不是表象或模糊不清的命名。

应该优先虑以这元素具体目来进行命名，这样他就最容易理解，减少更新。

通用名称可以加在兄弟元素都不特殊或没有个别意义的元素上，可以起名类似“helpers”这样的泛。

使用功能性或通用的名字会减少不必要的文档或模板修改。

```
/* 不推荐: 无意义 不易理解 */

#yee-1901 {}


/* 不推荐: 表达不具体 */

.button-green {}

.clear {}

/* 推荐: 明确详细 */

#gallery {}

#login {}

.video {}


/* 推荐: 通用 */

.aux {}

.alt {}
```

ID 和 class 命名风格

非必要的情况下，ID 和 class 的名称应尽量简短。

简要传达 ID 或 class 是关于什么的。

通过这种方式，似的代码易懂且高效。

```
/* 不推荐 */
```

```
#navigation {}

.atr {}

/* 推荐 */

#nav {}

.author {}
```

类型选择器

避免使用 CSS 类型选择器。

非必要的情况下不要使用元素标签名和 ID 或 class 进行组合。

出于性能上的考虑避免使用父辈节点做选择器 performance reasons.

```
/* 不推荐 */

ul#example {}

div.error {}

/* 推荐 */

#example {}

.error {}
```

属性缩写

写属性值的时候尽量使用缩写。

CSS 很多属性都支持缩写 shorthand （例如 font ） 尽量使用缩写，甚至只设置一个值。

使用缩写可以提高代码的效率和方便理解。

```
/* 不推荐 */

border-top-style: none;
```

```
font-family: palatino, georgia, serif;

font-size: 100%;

line-height: 1.6;

padding-bottom: 2em;

padding-left: 1em;

padding-right: 1em;

padding-top: 0;

/* 推荐 */

border-top: 0;

font: 100%/1.6 palatino, georgia, serif;

padding: 0 1em 2em;
```

0 和单位

省略 0 后面的单位。

非必要的情况下 0 后面不用加单位。

```
margin: 0;

padding: 0;
```

0 开头的小数

省略 0 开头小数点前面的 0。

值或长度在-1 与 1 之间的小数，小数前的 0 可以忽略不写。

```
font-size: .8em;
```

URI 外的引号

省略 URI 外的引号。

不要在 url() 里用（ ″″ ， ’’ ）。

@import url(//www.google.com/css/go.css);

十六进制

十六进制尽可能使用 3 个字符。

加颜色值时候会用到它，使用 3 个字符的十六进制更短与简洁。

/* 不推荐 */

color: #eebbcc;

/* 推荐 */

color: #ebc;

前缀

选择器前面加上特殊应用标识的前缀（可选）。

大型项目中最好在 ID 或 class 名字前加上这种标识性前缀（命名空间），使用短破折号链接。

使用命名空间可以防止命名冲突，方便维护，比如在搜索和替换操作上。

.adw-help {} /* AdWords */

#maia-note {} /* Maia */

ID 和 class 命名的定界符

ID 和 class 名字有多单词组合的用短破折号“-”分开。

别在选择器名字里用短破折号“-”以外的连接词(包括啥也没有)， 以增进对名字的理解和查找。

/* 不推荐：“demo”和“image”中间没加“-” */

.demoimage {}

```
/* 不推荐：用下划线“_”是屌丝的风格 */

.error_status {}

/* 推荐 */

#video-id {}

.ads-sample {}
```

Hacks

最好避免使用该死的 CSS “hacks” —— 请先尝试使用其他的解决方法。

虽然它很有诱惑力，可以当作用户代理检测或特殊的 CSS 过滤器，但它的行为太过于频繁，会长期伤害项目的效率和代码管理，所以能用其他的解决方案就找其他的。

CSS 代码格式规则

声明顺序

依字母顺序进行声明。

都按字母顺序声明，很容易记住和维护。

忽略浏览器的特定前缀排序，但多浏览器特定的某个 CSS 属性前缀应相对保持排序（例如-moz 前缀在-webkit 前面）。

```
background: fuchsia;

border: 1px solid;

-moz-border-radius: 4px;

-webkit-border-radius: 4px;

border-radius: 4px;

color: black;
```

```
text-align: center;

text-indent: 2em;
```

代码块内容缩进

缩进所有代码块（“{}”之间）内容。

缩进所有代码块的内容，它能够提高层次结构的清晰度。

```
@media screen, projection {


  html {

    background: #fff;

    color: #444;

  }


}
```

声明完结

所有声明都要用“;”结尾。

考虑到一致性和拓展性，请在每个声明尾部都加上分号。

```
/* 不推荐 */

.test {

  display: block;

  height: 100px

}

/* 推荐 */
```

```
.test {
  display: block;
  height: 100px;
}
```

属性名完结

在属性名冒号结束后加一个空字符。

出于一致性的原因，在属性名和值之间加一个空格（可不是属性名和冒号之间噢）。

```
/* 不推荐 */
h3 {
  font-weight:bold;
}
/* 推荐 */
h3 {
  font-weight: bold;
}
```

选择器和声明分行

将选择器和声明隔行。

每个选择器和声明都要独立新行。

```
/* 不推荐 */
a:focus, a:active {
  position: relative; top: 1px;
```

```
}

/* 推荐 */

h1,

h2,

h3 {

  font-weight: normal;

  line-height: 1.2;

}
```

规则分行

每个规则独立一行。

两个规则之间隔行。

```
html {

  background: #fff;

}


body {

  margin: auto;

  width: 50%;

}
```

CSS 元数据规则

注释部分

按组写注释。（可选）

如果可以，按照功能的类别来对一组样式表写统一注释。独立成行。

```
/* Header */


#adw-header {}


/* Footer */


#adw-footer {}


/* Gallery */


.adw-gallery {}
```