



加入伯乐在线专栏作者

扩大知名度，还能得 **赞赏**



[首页](#)
[最新文章](#)
[经典回顾](#)
[开发](#)
[设计](#)
[极客](#)
[IT技术](#)
[业界](#)
[职场](#)
[创业](#)
[访谈](#)
[在国外](#)

- 导航条 - ▼

[伯乐在线](#) > [首页](#) > [所有文章](#) > [Python](#) > 深刻理解Python中的元类(metaclass)

深刻理解Python中的元类(metaclass)

2014/08/10 • [Python](#), [开发](#) • [19 评论](#) • [metaclass](#), [Python](#)

分享到： **163** 本文由 [伯乐在线](#) - [bigship](#) 翻译。未经许可，禁止转载！
英文出处：[stackoverflow](#)。欢迎加入[翻译组](#)。

译注：这是一篇在[Stack overflow](#)上很热的帖子。提问者自称已经掌握了有关Python OOP编程中的各种概念，但始终觉得元类(metaclass)难以理解。他知道这肯定和自省有关，但仍然觉得不太明白，希望大家可以给出一些实际的例子和代码片段以帮助理解，以及在什么情况下需要进行元编程。于是e-satis同学给出了神一般的回复，该回复获得了985点的赞同点数，更有人评论说这段回复应该加入到Python的官方文档中去。而e-satis同学本人在Stack Overflow中的声望积分也高达64271分。以下就是这篇精彩的回复（提示：非常长）



类也是对象

在理解元类之前，你需要先掌握Python中的类。Python中类的概念借鉴于Smalltalk，这显得有些奇

特。在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成立。

首页 资讯 文章 频道 ▾ 资源 小组 ♡ 相亲

频道 ▾ 登录 注册 ? Python

```
1 >>> class ObjectCreator(object):
2 ...     pass
3 ...
4 >>> my_object = ObjectCreator()
5 >>> print my_object
6 <__main__.ObjectCreator object at 0x8974f2c>
```

但是，Python中的类还远不止如此。类同样也是一种对象。是的，没错，就是对象。只要你使用关键字class，Python解释器在执行的时候就会创建一个对象。下面的代码段：

Python

```
1 >>> class ObjectCreator(object):
2 ...     pass
3 ...
```

将在内存中创建一个对象，名字就是ObjectCreator。这个对象（类）自身拥有创建对象（类实例）的能力，而这就是为什么它是一个类的原因。但是，它的本质仍然是一个对象，于是乎你可以对它做如下的操作：

- 1) 你可以将它赋值给一个变量
- 2) 你可以拷贝它
- 3) 你可以为它增加属性
- 4) 你可以将它作为函数参数进行传递

下面是示例：

Python

```
1 >>> print ObjectCreator # 你可以打印一个类，因为它其实也是一个对象
2 <class '__main__.ObjectCreator'>
3 >>> def echo(o):
4 ...     print o
5 ...
6 >>> echo(ObjectCreator) # 你可以将类做为参数传给函数
7 <class '__main__.ObjectCreator'>
8 >>> print hasattr(ObjectCreator, 'new_attribute')
9 False
10 >>> ObjectCreator.new_attribute = 'foo' # 你可以为类增加属性
11 >>> print hasattr(ObjectCreator, 'new_attribute')
12 True
13 >>> print ObjectCreator.new_attribute
14 foo
15 >>> ObjectCreatorMirror = ObjectCreator # 你可以将类赋值给一个变量
16 >>> print ObjectCreatorMirror()
17 <__main__.ObjectCreator object at 0x8997b4c>
```

动态地创建类

因为类也是对象，你可以在运行时动态的创建它们，就像其他任何对象一样。首先，你可以在函数中创建类，使用class关键字即可。

Python

```
1 >>> def choose_class(name):
```

```

2 ...     if name == 'foo':
3 ...         class Foo(object):
4 ...             pass
5 ...         return Foo      # 返回的是类，不是类的实例
6 ...     else:
7 ...         class Bar(object):
8 ...             pass
9 ...         return Bar
10 ...
11 >>> MyClass = choose_class('foo')
12 >>> print MyClass          # 函数返回的是类，不是类的实例
13 <class '__main__.Foo'>
14 >>> print MyClass()        # 你可以通过这个类创建类实例，也就是对象
15 <__main__.Foo object at 0x89c6d4c>

```

但这还不够动态，因为你仍然需要自己编写整个类的代码。由于类也是对象，所以它们必须是通过什么东西来生成的才对。当你使用class关键字时，Python解释器自动创建这个对象。但就和Python中的大多数事情一样，Python仍然提供给你手动处理的方法。还记得内建函数type吗？这个古老但强大的函数能够让你知道一个对象的类型是什么，就像这样：

Python

```

1 >>> print type(1)
2 <type 'int'>
3 >>> print type("1")
4 <type 'str'>
5 >>> print type(ObjectCreator)
6 <type 'type'>
7 >>> print type(ObjectCreator())
8 <class '__main__.ObjectCreator'>

```

这里，type有一种完全不同的能力，它也能动态的创建类。type可以接受一个类的描述作为参数，然后返回一个类。（我知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在Python中是为了保持向后兼容性）

type可以像这样工作：

Python

```

1 type(类名, 父类的元组（针对继承的情况，可以为空），包含属性的字典（名称和值））

```

比如下面的代码：

Python

```

1 >>> class MyShinyClass(object):
2 ...     pass

```

可以手动像这样创建：

Python

```

1 >>> MyShinyClass = type('MyShinyClass', (), {}) # 返回一个类对象
2 >>> print MyShinyClass
3 <class '__main__.MyShinyClass'>
4 >>> print MyShinyClass() # 创建一个该类的实例
5 <__main__.MyShinyClass object at 0x8997cec>

```

你会发现我们使用“MyShinyClass”作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。

type 接受一个字典来为类定义属性，因此

Python

```
1 >>> class Foo(object):
2 ...     bar = True
```

可以翻译为:

Python

```
1 >>> Foo = type('Foo', (), {'bar':True})
```

并且可以将Foo当成一个普通的类一样使用:

Python

```
1 >>> print Foo
2 <class '__main__.Foo'>
3 >>> print Foo.bar
4 True
5 >>> f = Foo()
6 >>> print f
7 <__main__.Foo object at 0x8a9b84c>
8 >>> print f.bar
9 True
```

当然, 你可以向这个类继承, 所以, 如下的代码:

Python

```
1 >>> class FooChild(Foo):
2 ...     pass
```

就可以写成:

Python

```
1 >>> FooChild = type('FooChild', (Foo,), {})
2 >>> print FooChild
3 <class '__main__.FooChild'>
4 >>> print FooChild.bar    # bar属性是由Foo继承而来
5 True
```

最终你会希望为你的类增加方法。只需要定义一个有着恰当签名的函数并将其作为属性赋值就可以了。

Python

```
1 >>> def echo_bar(self):
2 ...     print self.bar
3 ...
4 >>> FooChild = type('FooChild', (Foo,), {'echo_bar': echo_bar})
5 >>> hasattr(Foo, 'echo_bar')
6 False
7 >>> hasattr(FooChild, 'echo_bar')
8 True
9 >>> my_foo = FooChild()
10 >>> my_foo.echo_bar()
11 True
```

你可以看到, 在Python中, 类也是对象, 你可以动态的创建类。这就是当你使用关键字class时Python在幕后做的事情, 而这就是通过元类来实现的。

到底什么是元类 (终于到主题了)

元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象, 不是吗? 但是我们已经学习到

了Python中的类也是对象。好吧，元类就是用来创建这些类（对象）的，元类就是类的类，你可以这样理解 为：

Python

```
1 MyClass = MetaClass()
2 MyObject = MyClass()
```

你已经看到了type可以让你像这样做：

Python

```
1 MyClass = type('MyClass', (), {})
```

这是因为函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。现在你想知道那为什么type会全部采用小写形式而不是Type呢？好吧，我猜这是为了和str保持一致性，str是用来创建字符串对象的类，而int是用来创建整数对象的类。type就是创建类对象的类。你可以通过检查__class__属性来看到这一点。Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来。

Python

```
1 >>> age = 35
2 >>> age.__class__
3 <type 'int'>
4 >>> name = 'bob'
5 >>> name.__class__
6 <type 'str'>
7 >>> def foo(): pass
8 >>> foo.__class__
9 <type 'function'>
10 >>> class Bar(object): pass
11 >>> b = Bar()
12 >>> b.__class__
13 <class '__main__.Bar'>
```

现在，对于任何一个__class__的__class__属性又是什么呢？

Python

```
1 >>> a.__class__.__class__
2 <type 'type'>
3 >>> age.__class__.__class__
4 <type 'type'>
5 >>> foo.__class__.__class__
6 <type 'type'>
7 >>> b.__class__.__class__
8 <type 'type'>
```

因此，元类就是创建类这种对象的东西。如果你喜欢的话，可以把元类称为“类工厂”（不要和工厂类搞混了:D） type就是Python的内建元类，当然了，你也可以创建自己的元类。

__metaclass__属性

你可以在写一个类的时候为其添加__metaclass__属性。

Python

```
1 class Foo(object):
2     __metaclass__ = something...
3 [...]
```

如果你这么做了，Python就会用元类来创建类Foo。小心点，这里面有些技巧。你首先写下class Foo(object)，但是类对象Foo还没有在内存中创建。Python会在类的定义中寻找__metaclass__属性，如果找到了，Python就会用它来创建类Foo，如果没有找到，就会用内建的type来创建这个类。把下面这段话反复读几次。当你写如下代码时：

```
Python
1 class Foo(Bar):
2     pass
```

Python做了如下的操作：

Foo中有__metaclass__这个属性吗？如果是，Python会在内存中通过__metaclass__创建一个名字为Foo的类对象（我说的是类对象，请紧跟我的思路）。如果Python没有找到__metaclass__，它会继续在Bar（父类）中寻找__metaclass__属性，并尝试做和前面同样的操作。如果Python在任何父类中都找不到__metaclass__，它就会在模块层次中去寻找__metaclass__，并尝试做同样的操作。如果还是找不到__metaclass__，Python就会用内置的type来创建这个类对象。

现在的问题就是，你可以在__metaclass__中放置些什么代码呢？答案就是：可以创建一个类的东西。那么什么可以用来创建一个类呢？type，或者任何使用到type或者子类化type的东东都可以。

自定义元类

元类的主要目的就是为了当创建类时能够自动地改变类。通常，你会为API做这样的事情，你希望可以创建符合当前上下文的类。假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定__metaclass__。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，__metaclass__实际上可以被任意调用，它并不需要是一个正式的类（我知道，某些名字里带有‘class’的东西并不需要是一个class，画画图理解下，这很有帮助）。所以，我们这里就先以一个简单的函数作为例子开始。

```
Python
1 # 元类会自动将你通常传给‘type’的参数作为自己的参数传入
2 def upper_attr(future_class_name, future_class_parents, future_class_attr):
3     '''返回一个类对象，将属性都转为大写形式'''
4     # 选择所有不以'__'开头的属性
5     attrs = ((name, value) for name, value in future_class_attr.items() if not name.startswith('__'))
```

```
Python
1     # 将它们转为大写形式
2     uppercase_attr = dict((name.upper(), value) for name, value in attrs)
3
4     # 通过'type'来做类对象的创建
5     return type(future_class_name, future_class_parents, uppercase_attr)
6
7 __metaclass__ = upper_attr # 这会作用到这个模块中的所有类
8
9 class Foo(object):
10     # 我们也可以只在这里定义__metaclass__，这样就只会作用于这个类中
11     bar = 'bip'
```

```
Python
1 print hasattr(Foo, 'bar')
2 # 输出: False
3 print hasattr(Foo, 'BAR')
4 # 输出: True
5
```



```

6 f = Foo()
7 print f.BAR
8 # 输出:'bip'

```

现在让我们再做一次，这一次用一个真正的class来当做元类。

Python

```

1 # 请记住，'type'实际上是一个类，就像'str'和'int'一样
2 # 所以，你可以从type继承
3 class UpperAttrMetaClass(type):
4     # __new__ 是在__init__之前被调用的特殊方法
5     # __new__是用来创建对象并返回之的方法
6     # 而__init__只是用来将传入的参数初始化给对象
7     # 你很少用到__new__，除非你希望能够控制对象的创建
8     # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
9     # 如果你希望的话，你也可以在__init__中做些事情
10    # 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
11    def __new__(upperattr_metaclass, future_class_name, future_class_parents, future_c
12                attrs = ((name, value) for name, value in future_class_attr.items() if not na
13                uppercase_attr = dict((name.upper(), value) for name, value in attrs)
14    return type(future_class_name, future_class_parents, uppercase_attr)

```

但是，这种方式其实不是OOP。我们直接调用了type，而且我们没有改写父类的__new__方法。现在让我们这样去处理：

Python

```

1 class UpperAttrMetaClass(type):
2     def __new__(upperattr_metaclass, future_class_name, future_class_parents, future_c
3         attrs = ((name, value) for name, value in future_class_attr.items() if not na
4         uppercase_attr = dict((name.upper(), value) for name, value in attrs)
5
6         # 复用type.__new__方法
7         # 这就是基本的OOP编程，没什么魔法
8         return type.__new__(upperattr_metaclass, future_class_name, future_class_pare

```

你可能已经注意到了有个额外的参数upperattr_metaclass，这并没有什么特别的。类方法的第一个参数总是表示当前的实例，就像在普通的类方法中的self参数一样。当然了，为了清晰起见，这里的名字我起的比较长。但是就像self一样，所有的参数都有它们的传统名称。因此，在真实的产品代码中一个元类应该是像这样的：

Python

```

1 class UpperAttrMetaClass(type):
2     def __new__(cls, name, bases, dct):
3         attrs = ((name, value) for name, value in dct.items() if not name.startswith(
4         uppercase_attr = dict((name.upper(), value) for name, value in attrs)
5         return type.__new__(cls, name, bases, uppercase_attr)

```

如果使用super方法的话，我们还可以使它变得更清晰一些，这会缓解继承（是的，你可以拥有元类，从元类继承，从type继承）

Python

```

1 class UpperAttrMetaClass(type):
2     def __new__(cls, name, bases, dct):
3         attrs = ((name, value) for name, value in dct.items() if not name.startswith(
4         uppercase_attr = dict((name.upper(), value) for name, value in attrs)
5         return super(UpperAttrMetaClass, cls).__new__(cls, name, bases, uppercase_att

```

就是这样，除此之外，关于元类真的没有别的可说的了。使用到元类的代码比较复杂，这背后的原因倒并不是因为元类本身，而是因为你通常会使用元类去做一些晦涩的事情，依赖于自省，控制继承等等。

确实，用元类来搞些“黑暗魔法”是特别有用的，因而会搞出些复杂的东西来。但就元类本身而言，它们其实是很简单的：

- 1) 拦截类的创建
- 2) 修改类
- 3) 返回修改之后的类

为什么要用metaclass类而不是函数？

由于__metaclass__可以接受任何可调用的对象，那为何还要使用类呢，因为很显然使用类会更加复杂啊？这里有好几个原因：

- 1) 意图会更加清晰。当你读到UpperAttrMetaclass(type)时，你知道接下来要发生什么。
- 2) 你可以使用OOP编程。元类可以从元类中继承而来，改写父类的方法。元类甚至还可以使用元类。
- 3) 你可以把代码组织的更好。当你使用元类的时候肯定不会是像我上面举的这种简单场景，通常都是针对比较复杂的问题。将多个方法归总到一个类中会很有帮助，也会使得代码更容易阅读。
- 4) 你可以使用__new__，__init__以及__call__这样的特殊方法。它们能帮你处理不同的任务。就算通常你可以把所有的东西都在__new__里处理掉，有些人还是觉得用__init__更舒服些。
- 5) 哇哦，这东西的名字是metaclass，肯定非善类，我要小心！

究竟为什么要使用元类？

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？好吧，一般来说，你根本就用不上它：

“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。” —— Python界的领袖 Tim Peters

元类的主要用途是创建API。一个典型的例子是Django ORM。它允许你像这样定义：

```
Python
1 class Person(models.Model):
2     name = models.CharField(max_length=30)
3     age = models.IntegerField()
```

但是如果你像这样做的话：

```
Python
1 guy = Person(name='bob', age='35')
2 print guy.age
```

这并不会返回一个IntegerField对象，而是会返回一个int，甚至可以直接从数据库中取出数据。这是有可能的，因为models.Model定义了__metaclass__，并且使用了一些魔法能够将你刚刚定义的简单的Person类转变成对数据库的一个复杂hook。Django框架将这些看起来很复杂的东西通过暴露出一个简单的使用元类的API将其化简，通过这个API重新创建代码，在背后完成真正的工作。

结语

首先，你知道了类其实是能够创建出类实例的对象。好吧，事实上，类本身也是实例，当然，它们是元类的实例。

Python

```
1 >>>class Foo(object): pass
2 >>> id(Foo)
3 142630324
```

Python中的一切都是对象，它们要么是类的实例，要么是元类的实例，除了type。type实际上是它自己的元类，在纯Python环境中这可不是你能够做到的，这是通过在实现层面耍一些小手段做到的。其次，元类是很复杂的。对于非常简单的类，你可能不希望通过使用元类来对类做修改。你可以通过其他两种技术来修改类：

1) [Monkey patching](#)

2) class decorators

当你需要动态修改类时，99%的时间里你最好使用上面这两种技术。当然了，其实在99%的时间里你根本就不需要动态修改类 :D

拿高薪，还能扩大业界知名度！优秀的开发工程师看过来 -> 《[高薪招募讲师](#)》

👍 1 赞

🔖 18 收藏

💬 19 评论

关于作者：bigship



简介还没来得及写 :) [👤 个人主页](#) · [📄 我的文章](#) · [🎓 4](#)



相关文章

- [Python高级特性（3）：Classes和Metaclasses](#)
- [Python十分钟入门](#)
- [Python程序员的10个常见错误](#)
- [Python 新手常犯错误（第一部分）](#)
- [Python 新手常犯错误（第二部分）](#)
- [一起来写个简单的解释器（6）](#)
- [12岁的少年教你用Python做小游戏](#)
- [Python编程中常用的12种基础知识总结](#)
- [我常用的 Python 调试工具](#)
- [Python高级编程技巧](#)

可能感兴趣的话题

- [腾讯2015春招移动客户端开发练习卷\(android\)](#) · [💬 3](#)
- [你最喜欢的粤语歌，求推荐](#) · [💬 68](#)
- [一个图片上传相关的 JS 问题求助](#) · [💬 18](#)
- [想学Web前端的小白，困惑多多，求大家帮忙解答](#) · [💬 31](#)
- [2016京东在线笔试题\(java\)](#) · [💬 7](#)

- [摄像头识别激光光线，实时判断是否为直线，求助应该用什么？](#) • [🗨️ 4](#)
- [一枚小小的实习生要不要跳槽呢？](#) • [🗨️ 5](#)
- [Python 匹配指定中文词](#)
- [有组队一起去看这周五上映的《谁的青春不迷茫》的吗](#) • [🗨️ 23](#)
- [2017蘑菇街校招笔试题\(Servlet问题\)](#) • [🗨️ 7](#)

« [老码农冒死揭开行业黑幕：如何编写无法维护的代码](#) [Android每周热点第二十六期](#) »

[登录后评论](#)[新用户注册](#)

直接登录



最新评论



酿泉

[2012/12/24](#)

非常感谢

[👍 赞](#) [🗨️ 回复](#) [↩️](#)

ctrlaltdel1

[2013/04/14](#)

很棒~

[👍 赞](#) [🗨️ 回复](#) [↩️](#)

yue

[2013/05/02](#)

太棒了，非常感谢，我想转载

[👍 赞](#) [🗨️ 回复](#) [↩️](#)

千年湖之心

[2013/07/23](#)

学习了，元类看起来很强大。目前看起来更像一个额外知识库，希望以后用得上

[👍 赞](#) [🗨️ 回复](#) [↩️](#)

August Trek

[2013/11/25](#)

「根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情」为什么说是很傻的事情？

[👍 赞](#) [🗨️ 回复](#) [↩️](#)[bigship](#) ([🎓 4](#))[2013/11/26](#)

函数的基本原则：do one thing, do it well .

[👍 赞](#) [🗨️ 回复](#) [↩️](#)



[leoliu](#) (1)

[2014/05/16](#)

如果使用super方法的话，我们还可以使它变得更清晰一些，这会缓解继承（是的，你可以拥有元类，从元类继承，从type继承）

这里的缓解继承是什么意思？

赞 [回复](#)



Python小菜

[2014/07/02](#)

在Python3.4里怎么不行？？Python2.x和Python3.x的元类有什么区别吗？？检查了几遍貌似没抄错

赞 [回复](#)



[lix](#)

[2014/08/27](#)

3.X里的元类要这样用了：
`class C(metaclass=M):`
...

赞 [回复](#)



,mzfe

[2014/11/06](#)

go figure不是“画画图”的意思。翻译过的文章读起来比原文还费劲。

赞 [回复](#)



k9

[2014/11/28](#)

哥们，不要这样，人家翻译得不错，整篇文章都可以

赞 [回复](#)



[dongguangming](#) (1)

[2014/12/09](#)

好文章。

赞 [回复](#)



yiyuezhao

[2015/02/24](#)

这篇文章问了两遍，第一次看的时候昏昏沉沉觉得用处不大就弃了。这次想一种简洁的定义方法如何实现，左思右想不得其解(包括装饰器，猴子补丁之流)，最后终于想起有个叫元类的东西，一看果然是我要的效果。这篇文章真是太棒了。

赞 [回复](#)



[infinity1207](#) (1 ·)

[2015/07/15](#)

翻译的很不错，看完终于把元类搞清楚了，这样的文章看的真过瘾。

赞 [回复](#)



Hello

2015/10/21

这篇文章好~非常易懂~

[👍 赞](#) [回复](#) [↩](#)kewinwang (1)
SE

01/16

自定义元类的实例中的 `class Foo(object)` 是错误的 代码会报错 应该是`class Foo()` stackoverflow中有解释

[👍 赞](#) [回复](#) [↩](#)

洛荷 (1 ·)

01/25

自定义元类不能从'object'继承, metaclass是类的模板, 所以必须从`type`类型派生。可以这么定义一个元类`class foo(type)`, 然后这么用 `class myfoo(metaclass=foo)`、

[👍 赞](#) [回复](#) [↩](#)

夏洛之枫 (1)

01/27

翻译有错误! 使用函数作为`__metaclass__`时, 接下定义的类不能继承自object, 我看了原文才知道, 原作者写的类是`class Foo():`, 而翻译成了`class Foo(object)`, 这是不对的

[👍 1 赞](#) [回复](#) [↩](#)

Asgard.Golua (1)

6 天前

翻译的有问题, 很多重要的话没翻译清楚, 会误导新手的, 例如You may have noticed the extra argument upperattr_metaclass. There is nothing special about it: `__new__` always receives the class it's defined in, as first parameter. Just like you have self for ordinary methods which receive the instance as first parameter, or the defining class for class methods.

```
# global __metaclass__ won't work with "object" though
# but we can define __metaclass__ here instead to affect only this class
# and this will work with "object" children
```

等等, 这些概念不能随意翻译, 一点要准确。

[👍 赞](#) [回复](#) [↩](#)

文章 ▼

输入搜索关键字

搜索

- [本周热门文章](#)
- [本月热门文章](#)
- [热门标签](#)

0 [如何准备阿里社招面试?](#)

1 [趣文: 我是一个线程](#)

2 [算法的力量, 李开复聊算法的重要性](#)

3 [提高效率, 推荐 5 款命令行工具](#)

4 [程序员的骄傲, 以及骄傲背后真实的原因](#)

5 [聊聊IO多路复用之select、poll、epo...](#)

6 [聊聊 Linux 中的五种 IO 模型](#)

7 [为什么未来是全栈工程师的世界？](#)

8 [聊聊 C10K 问题及解决方案](#)

9 [面试中的排序算法总结](#)



业界热点资讯

[更多 >>](#)

[Open365: Microsoft Office 365 的开源替代品](#)

21 小时前 • 4



[谷歌开始为Chrome浏览器及系统部署Material Design](#)

2 天前 • 12



[2015年十大最危险漏洞：全部出自Adobe Flash](#)

1 天前 • 5



[神奇柔性可穿戴设备：通过分析汗液确定人体健康状况](#)

19 小时前 • 2

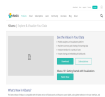


[Chromebook也许很快就能运行所有的Android应用](#)

1 天前 • 4 • 1



精选工具资源

[更多资源 >>](#)

[Kibana: 分析及可视化日志文件
日志](#)



[Reactor: 构建响应式快速数据应用程序的开发库
响应式开发库](#)



[Logback: 旨在取代 log4j 的日志组件
日志](#)



[JavaMelody: 性能监测和分析工具
应用监控工具](#)



[HK2: 轻量级动态依赖注入框架
依赖注入](#)



最新评论



Re: [算法的力量，李开复聊算法的重要性](#)

我要站出来说句话了，文中对于算法的作用过于夸大了，虽然文中所说的都没错，但其实作者依然是在拿特定的问...

-  Re: [狼、兔子，人性才是最重要的](#)
管理是有政治陷阱的，技术也是有政治陷阱的。认清自己所处的环境才是最难的。
-  Re: [蜕变成蝶：Linux设备驱动中的阻...](#)
需要点基础。
-  Re: [Web 应用性能提升 10 倍的 1...](#)
nginx线程池？这个在哪边配置。
-  Re: [Linux Shell 脚本实现 tcp/u...](#)
有好多shell的文章，不错。
-  Re: [优秀程序员的这些秘诀，你知道几...](#)
浅显易懂小习惯可以改变大世界
-  Re: [如何准备阿里社招面试？](#)
现在只有比较传统的公司和一些老的项目才用hibernate，互联网和一些新的项目都是用的mybati...
-  Re: [趣文：我是一个线程](#)
非常形象，好生动哈，大学的时候老师如果这么讲课该多好。

关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为“快餐”添加一些“营养”元素。

快速链接

[问题反馈与求助](#) »

[加入伯乐翻译小组](#) »

[加入专栏作者](#) »

关注我们

新浪微博: [@伯乐在线官方微博](#)

RSS: [订阅地址](#)

推荐微信号



程序员的那些事



UI设计达人



极客范


合作联系

Email: bd@jobbole.com

QQ: 2302462408 (加好友请注明来意)

更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
 - [头条](#) - 分享和发现有价值的内容与观点
 - [相亲](#) - 为IT单身男女服务的征婚传播平台
 - [资源](#) - 优秀的工具资源导航
 - [翻译](#) - 翻译传播优秀的外文文章
 - [文章](#) - 国内外的精选文章
 - [设计](#) - UI, 网页, 交互和用户体验
 - [iOS](#) - 专注iOS技术分享
 - [安卓](#) - 专注Android技术分享
 - [前端](#) - JavaScript, HTML5, CSS
 - [Java](#) - 专注Java技术分享
 - [Python](#) - 专注Python技术分享
-

© 2016 伯乐在线 [首页](#) [博客](#) [资源](#) [小组](#) [相亲](#)  [反馈](#)

