开发人员指南

欢迎使用 Google App Engine！本开发人员指南包含您使用 Google 技术构建可扩展网络应用程序需要了解的所有内容。

本指南包含以下小节：

- 简介
  - 什么是 Google Engine？，对 App Engine 的功能和服务的简介
  - 使用入门，用工作实例介绍 App Engine API 的快速教程。如果您是第一次使用 App Engine，请从此处开始。
- API
  - Python 运行时，有关您的应用程序在其中运行的 Python 环境、沙盒功能、应用程序缓存、日志的内容
  - 数据库 API，所有有关可扩展数据库和如何有效使用的内容
  - 图像 API，图像数据处理服务
  - 邮件 API，从您的应用程序发送电子邮件
  - Memcache API，分布式内存缓存
  - 网址抓取 API，从您的应用程序访问其他互联网主机
  - 用户 API，将您的应用程序与 Google 帐户集成
- 工具和配置
  - Webapp 框架，网络应用程序的简单基础
  - 配置应用程序，所有有关 app.yaml 的内容
  - 配置索引，所有有关 index.yaml 的内容
  - 开发网络服务器，在您的计算机上用 dev_appserver.py 模拟 App Engine
  - 上传应用程序，用 appcfg.py 更新您的应用程序的公共版本
  - 管理控制台，管理和监控您的应用程序的访问量和日志，在主要版本间切换

什么是 Google App Engine？

Google App Engine 可让您在 Google 的基础架构上运行您的网络应用程序。App Engine 应用程序易于构建和维护，并可根据您的访问量和数据存储需要的增长轻松扩展。使用 Google App Engine，将不再需要维护服务器：您只需上传您的应用程序，它便可立即为您的用户提供服务。

您可以使用 appspot.com 域上的免费域名为您的应用程序提供服务，也可以使用 Google 企业应用套件从您自己的域为它提供服务。您可以与全世界的人共享您的应用程序，也可以限制为只有您组织内的成员可以访问。

可以免费开始使用 App Engine。注册一个免费帐户即可开发和发布您的应用程序以供全世界的人共享，而且不需要承担任何费用和责任。免费帐户可以使用多达 500MB 的持久存储空间，以及可支持每月约 500 万页面浏览量的超大 CPU 和带宽。

使用 Google App Engine 的预览版期间，仅提供免费帐户。您很快就能够购买其他的计算资源。

应用程序环境

通过 Google App Engine，即使在负载很重和数据量极大的情况下，也可以轻松构建能安全运行的应用程序。该环境包括以下特性：

- 动态网络服务，提供对常用网络技术的完全支持
- 持久存储空间，支持查询、分类和事务
- 自动扩展和负载平衡
- 用于对用户进行身份验证和使用 Google 帐户发送电子邮件的 API
- 一种功能完整的本地开发环境，可以在您的计算机上模拟 Google App Engine

Google App Engine 应用程序是使用 Python 编程语言实现的。该运行时环境包括完整 Python 语言和多数 Python 标准库。

目前，Google App Engine 仅支持 Python 语言，但是我们希望将来它可以支持更多语言。

沙盒

应用程序在安全环境中运行，该安全环境仅提供对基础操作系统的有限访问权限。这些限制让 App Engine 可以在多个服务器之间分发应用程序的网络请求，并可以启动和停止服务器以满足访问量需求。沙盒将您的应用程序隔离在它自己的安全可靠环境中，该环境与网络 服务器的硬件、操作系统和物理位置无关。

安全沙盒环境的限制示例包括：

- 应用程序只能通过提供的网址抓取以及电子邮件服务和 API 访问互联网中的其他计算机。其他计算机只能通过在标准端口上进行 HTTP（或 HTTPS）请求来连接至该应用程序。
- 应用程序无法向文件系统写入。应用程序只能读取通过应用程序代码上传的文件。该应用程序必须使用 App Engine 数据库存储所有在请求之间持续存在的数据。
- 应用程序代码仅在响应网络请求时运行，且必须在几秒钟内返回响应数据。请求处理程序不能在响应发送后产生子进程或执行代码。

Python 运行时环境

App Engine 提供了一个使用 Python 编程语言的运行时环境。将来的版本将考虑使用其他编程语言和运行时环境配置。

Python 运行时环境使用 Python 版本 2.5.2。

该环境包括 Python 标准库。当然，您无法调用违反了沙盒限制的库方法，例如尝试打开套接字或向文件写入。为了方便起见，其主要功能不受该运行时环境支持的标准库中的多个模块已被禁用，而导入这些模块的代码将抛出错误。

应用程序代码只能以 Python 编写。具有用 C 编写的扩展的代码不受支持。

Python 环境为数据库、Google 帐户、网址抓取和电子邮件服务提供了丰富的 Python API。App Engine 还提供了一

个称为 webapp 的简单 Python 网络应用程序框架，从而可以轻松开始构建应用程序。

为了方便起见， App Engine 还包括 Django 网络应用程序框架 0.96.1 版。请注意， App Engine 数据库不是某些 Django 组件必需的关系数据库。某些组件（例如 Django 模板引擎）按照文档化的程序工作，而其他组件则需要做更多工作。有关将 Django 与 App Engine 配合使用的提示，请参阅文章部分。

只要其他第三方库是使用纯 Python 实现的并且不需要任何不受支持的标准库模块，您就可以使用您的应用程序上传这些库。

有关 Python 运行时环境的详细信息，请参阅 Python 运行时环境。

数据库

App Engine 提供了一个强大的分布式数据存储服务，其中包含查询引擎和事务功能。就像分布式网络服务器随访问量增加一样，该分布式数据库也会随数据而增加。

该 App Engine 数据库与传统关系数据库不同。数据对象（或 [实体]）有一类和一组属性。查询可以检索按属性值过滤和分类的指定种类的实体。属性值可以是受支持的属性值类型中的任何一种。

数据库的 Python API 包括一个可以定义数据库实体结构的数据建模接口。数据模型可以指示属性值必须位于指定范围内，如果未指定值，还可以提供默认值。您的应用程序可以根据需要向数据提供或多或少的结构。

数据库使用乐观锁定进行并发控制。如果有其他进程尝试更新某实体，而同时该实体位于以固定次数进行重新尝试的事务中，此时该实体将更新。应用程序可以在一个事务中执行多项数据库操作（全部成功或者全部失败，从而确保数据的完整性）。

数据库通过其分布式网络使用 [实体组] 实现事务。一个事务操作一个组内的实体。同一组的实体存储在一起，以高效执行事务。应用程序可以在实体创建时将实体分配到组。

有关数据库的详细信息，请参阅数据库 API 参考。

Google 帐户

App Engine 包括用于与 Google 帐户集成的服务 API。应用程序使用户可以通过 Google 帐户登录，并可以访问与该帐户关联的电子邮件地址和可显示的名称。使用 Google 帐户使用户可以更快地开始使用您的应用程序，因为用户可以不需要创建新帐户了。Google 帐户还省去只为您的应用程序实现用户帐户系统的麻烦。

如果您的应用程序正在 Google 企业应用套件下运行，则它可以与您组织的成员和 Google 企业应用套件帐户成员使用相同的功能。

用户 API 还可告知应用程序当前用户是否是应用程序的注册管理员。这样便可以轻松实现您站点上仅管理员可访问的区域。

有关与 Google 帐户集成的详细信息，请参阅用户 API 参考。

App Engine 服务

App Engine 提供了多种服务，从而可让您在管理应用程序的同时执行常规操作。提供了以下 API 以访问这些服务：

网址抓取

应用程序可以使用 App Engine 的网址抓取服务访问互联网上的资源，例如网络服务或其他数据。网址抓取服务使用用于为许多其他 Google 产品检索网页的高速 Google 基础架构来检索网络资源。有关网址抓取服务的详细信息，请参阅网址抓取 API 参考。

邮件

应用程序可以使用 App Engine 的邮件服务发送电子邮件。邮件服务使用 Google 基础架构发送电子邮件。有关邮件服务的详细信息，请参阅邮件 API 参考。

Memcache

Memcache 服务为您的应用程序提供了高性能的内存键值缓存，您可通过应用程序的多个实例访问该缓存。Memcache 对于那些不需要数据库的持久性存储和事务功能的数据很有用，例如临时数据或从数据库复制到缓存以进行高速访问的数据。有关 Memcache 服务的详细信息，请参阅 Memcache API 参考。

图像操作

图像服务使您的应用程序可以对图像进行操作。使用该 API，您可以对 JPEG 和 PNG 格式的图像进行缩放、裁剪、旋转和翻转。有关图像操作服务的详细信息，请参阅图像 API 参考。

开发工作流程

App Engine 软件开发套件 (SDK) 包括可以在您的本地计算机上模拟所有 App Engine 服务的网络服务器应用程序。该 SDK 包括 App Engine 中的所有 API 和库。该网络服务器还可以模拟安全沙盒环境，包括检查是否存在禁用模块的导入以及对不允许访问的系统资源的尝试访问。

Python SDK 完全使用 Python 实现，可以在装有 Python 2.5 的任何平台上运行，包括 Windows、Mac OS X 和 Linux。您可以在 Python 网站上获得适用于您的系统的 Python。该 SDK 以 Zip 文件提供，对于 Windows 和 Mac OS X 还提供安装程序。

您可以在此处下载该 SDK。

该 SDK 还包括可将您的应用程序上传到 App Engine 的工具。创建了您的应用程序的代码、静态文件和配置文件后，即可运行该工具上传数据。该工具会提示您提供 Google 帐户电子邮件地址和密码。

构建已在 App Engine 上运行的应用程序的新主要发行版时，可以将新发行版作为新版本上传。在您改为使用新版本之前，旧版本可以继续为用户提供服务。可以在旧版本仍运行的同时在 App Engine 上测试新版本。

管理控制台是基于网络的接口，用于管理在 App Engine 上运行的应用程序。您可以使用它创建新应用程序、配置域名、更改您的应用程序当前的版本、检查访问权限和错误日志以及浏览应用程序数据库。

限额和限制

创建 App Engine 应用程序不仅简单，而且是免费的！您可以创建帐户，然后发布一个用户可以立即使用的应用程序，无需承担任何费用和责任。通过免费帐户发布的应用程序可使用多达 500MB 的存储空间和多达每月 500 万的页面浏览量。

在此次预览中，仅提供免费帐户。您很快就能够以有竞争力的市场价格购买其他的计算资源。预览期过后，免费帐户可继续使用。

在此次预览中，最多可注册 3 个应用程序。

应用程序资源限制（[限额]）会不断刷新。如果您的应用程序达到基于时间的限额（例如带宽），则该限额将以指定限制的比率立即开始刷新。固定限额（例如存储空间）只能通过降低使用量来缓和限制。

有些功能会施加与限额无关的限制，以保护系统的稳定性。例如，当调用某应用程序以为网络请求提供服务时，该应用程序必须在几秒钟内发出响应。如果该 应用程序花费的时间过长，则进程会被终止并且服务器将向用户返回错误代码。响应超时是动态的，如果请求处理程序经常达到其超时，则可以缩短请求超时以节省 资源。

服务限制的另一示例是查询返回的结果数。一个查询最多可返回 1,000 条结果。本该返回更多结果的查询只能返回该最大值。在这种情况下，执行这种查询的请求不可能在超时前返回请求，但限制仍存在以节省数据库上的资源。

试图破坏或滥用限额（例如同时在多个帐户上操作应用程序）违反服务条款，并可能导致应用程序被禁用或帐户关闭。

有关详细信息...

有关 Google App Engine 的详细信息：

- 观看 Campfire One 视频或阅读抄本。
- 观看 App Engine 演示视频。
- 下载 SDK、注册帐户，然后阅读《使用入门指南》并学习教程。
- 浏览应用程序库以获取使用 App Engine 构建的应用程序的示例。
- 查看剩余 App Engine 文档。

欢迎使用 Google App Engine！

Python 运行时环境

App Engine 应用程序是使用 Python 编程语言实现的。App Engine Python 运行时环境包括 Python 解释器的专用版本、标准 Python 库、App Engine 的库和 API，以及网络服务器层的标准接口。

有关 Python 的详细信息，请参阅 Python 网站和 Python 文档。

本章包括以下小节：

请求和 CGI

App Engine 在收到您的应用程序的网络请求时，会调用与此网址相对应的处理程序脚本（如应用程序的 app.yaml 配置文件中所述）。App Engine 会使用 CGI 标准将请求数据传递到处理程序，并接收响应。

网址映射和处理程序脚本

App Engine 在收到您的应用程序的网络请求时，会把该请求传送到网络服务器。该网络服务器会检查应用程序的 app.yaml 文件获取与请求的网址相匹配的网址映射，以确定哪个 Python 处理程序脚本会处理该请求。

App Engine 使用多个网络服务器运行您的应用程序，并自动调整它所使用的服务器数量以便可靠地处理请求。指定的请求可能会传送到任何服务器，而且可能不是处理先前来自同一用户的请求的服务器。

CGI

请求一旦被传送到一个服务器和相应的处理程序脚本，App Engine 就会调用此脚本。正如 CGI 标准中所述，服务器会把请求数据置于环境变量和标准输入流中。脚本会执行请求所需的操作，然后准备响应并将响应置于标准输出流上。

有关环境变量和输入流数据格式的详细信息，可以参考 CGI 文档。大多数应用程序会使用一个库来解析 CGI 请求以及返回 CGI 响应，例如，使用来自 Python 标准库的 cgi 模块 或知道 CGI 协议（例如 webapp）的网络框架。

以下示例处理程序脚本显示了用户浏览器上的消息。它会把标识消息类型和消息内容的 HTTP 标头打印到标准输出流中。

```
print "Content-Type: text/plain"
print ""
print "Hello, world!"
```

不支持流

App Engine 会收集处理程序脚本写入标准输出流的所有数据，然后等待该脚本退出。脚本退出后，所有输出数据会发送至用户。App Engine 不支持在退出处理程序前向用户的浏览器发送数据。

自动压缩响应

如果客户端发送带有指明该客户端可接受压缩（通过 gzip）内容的请求的 HTTP 标头，App Engine 会自动压缩该响应数据并附加相应的响应标头。Accept-Encoding 和 User-Agent 请求标头用于决定该客户端是否可以可靠地接收压缩响应。自定义客户端可通过指定 Accept-Encoding 和 User-Agent 标头（带有 [gzip] 值）强行压缩内容。

沙盒

App Engine 应用程序可同时在许多网络服务器上运行。任何网络请求都可转至任何网络服务器，并且来自同一用户的多个请求可由不同网络服务器处理。通过跨多个网络服务器分布，App Engine 可确保您的应用程序在同时为多个用户服务时保持其可用性。

要允许 App Engine 按此方式分布您的应用程序，该应用程序应在受限制的 [沙盒] 环境中运行。在这种环境中，该应用程序可执行代码；可存储和查询 App Engine 数据库中的数据；可使用 App Engine 邮件、网址抓取和用户服务；可检查用户的网络请求以及准备响应。

App Engine 应用程序无法：

- 向文件系统写入。应用程序必须使用 App Engine 数据库存储永久数据。允许从文件系统中读取，并且可使用与该应用程序一起上传的所有应用程序文件。（作为 [静态] 文件上传的文件不保留在文件系统中。）
- 打开套接字或直接访问另一主机。应用程序可使用 App Engine 网址抓取服务分别向端口 80 和 443 上的其他主机发出 HTTP 和 HTTPS 请求。
- 产生子进程或线程。必须在几秒钟内于单个进程中处理对应用程序的网络请求。响应时间很长的进程会被终止，以避免使网络服务器负载过重。
- 进行其他类型的系统调用（例如，信号）。

纯 Python

Python 运行时环境使用 Python 2.5。

适用于 Python 运行时环境的所有代码必须是纯 Python，且不包括任何 C 扩展程序或其他必须编译的代码。

该环境包括 Python 标准库。有些模块已被禁用，因为 App Engine 不支持其核心函数（例如，联网或写入到文件系统）。此外，os 模块可用，但其不支持的功能被禁用。尝试导入不支持的模块或使用不支持的功能会抛出异常。

标准库中的几个模块已替换掉，或已经过自定义，可以与 App Engine 配合使用。例如：

- cPickle 又名 pickle。不支持 cPickle 特定的功能。
- 列集为空。可以成功导入，但会无法使用。
- 同样，以下模块也为空：imp、ftplib、select、socket
- tempfile 被禁用，但对 TemporaryFile（又名 StringIO）除外。

- logging 可用，且强烈建议用户使用！请查看以下信息。

除了 Python 标准库和 App Engine 库之外，运行时环境还包括以下第三方库：

- Django 0.96.1
- WebOb 0.9
- PyYAML 3.05

您可以通过将代码置于您的应用程序目录中以将其他纯 Python 库添加到该应用程序中。如果您在应用程序目录中创建指向模块目录的符号链接，则 appcfg.py 会跟随此链接，并且将该模块添加到您的应用程序中。

Python 模块包括包含您的应用程序根目录（包含 app.yaml 文件的目录）的路径。可通过利用根目录中的路径来使用您在应用程序根目录中创建的模块。请务必在子目录中创建 __init__.py 文件，这样，Python 会把这些子目录识别为数据包。

请求和应用程序缓存

Python 运行时环境会在单个网络服务器上的请求之间对导入的模块进行缓存，类似于独立 Python 应用程序仅加载一次模块的方式（即使模块由多个文件导入）。如果处理程序脚本提供 main() 例行程序，则运行时环境也会缓存脚本。否则，就会对每个请求加载处理程序脚本。

应用程序缓存在响应时间方面有明显的优势。我们建议所有的应用程序都使用 main() 例行程序，如下所述。

缓存导入的内容

为了提高效率，网络服务器会将导入的模块保存在内存中，并且对于同一服务器上的相同应用程序的后续请求，就不再重新加载或重新评估这些模块。大多数模块不会初始化任何全局数据，或在导入时没有其他副作用，所以缓存它们不会更改应用程序的行为。

如果您的应用程序导入的模块取决于针对每个请求进行评估的模块，则应用程序必须调整该缓存行为。

以下示例演示了缓存导入的模块的方式。由于 mymodule 只对一个网络服务器导入一次，所以全局 mymodule.counter 只会对服务器提出的第一次请求初始化为 0。后续请求则使用来自前一个请求的值。

```
### mymodule.py
counter= 0
def increment():
  global counter
    counter+= 1
  return counter


### myhandler.py
import mymodule
```

```
print "Content-Type: text/plain"
print ""
print "My number: " + str(mymodule.increment())
```

这会输出 My number: #，其中 # 是处理请求的网络服务器调用该处理程序的次数。

处理程序脚本也可以进行缓存

您可以让 App Engine 除缓存导入模块之外，还对处理程序脚本本身进行缓存。如果处理程序脚本定义了一个名为 main() 的函数，则会缓存脚本及其全局环境，就像缓存导入的模块一样。指定网络服务器上脚本的第一个请求会正常评估脚本。对于后续请求，App Engine 则会调用缓存的环境中的 main() 函数。

要缓存处理程序脚本，App Engine 必须能够调用不带参数的 main()。如果处理程序脚本没有定义 main() 函数，或 main() 函数需要参数（没有默认值），则 App Engine 将针对每个请求加载和评估整个脚本。

将解析的 Python 代码保留在内存中可节省时间并加快响应的速度。缓存全局环境也有其他潜在作用：

- 编译的正则表达式。所有正则表达式都以编译的形式进行解析和存储。您可以在全局变量中存储编译的正则表达式，然后使用应用程序缓存以在请求之间重复使用编译的对象。
- GqlQuery 对象。创建 GqlQuery 对象时，会解析 GQL 查询字符串。重复使用具有参数绑定和 bind() 方法的 GqlQuery 对象要比每次都重新构建对象更快。您可为全局变量中的值存储具有参数绑定的 GqlQuery 对象，然后通过对每个请求绑定新参数值来重复使用该对象。
- 配置和数据文件。如果您的应用程序加载和解析来自文件的配置数据，它可以在内存中保留解析的数据，以免对每个请求都重新加载文件。

以下示例使用处理程序脚本的全局环境的缓存，可实现与前面的示例相同的操作：

```
### myhandler.py

# A global variable, cached between requests on this web server.
counter= 0

def main():
  global counter
    counter+= 1
  print "Content-Type: text/plain"
  print ""
  print "My number: " + str(counter)

if __name__== "__main__":
    main()
```

注意：请勿在请求之间 [泄漏] 用户特定的信息。除非需要缓存，否则请避免使用全局变量，且一律在 main() 例行

程序内初始化请求特定的数据。

具有 main() 的应用程序缓存在应用程序的响应时间方面有明显改善。我们建议将其用于所有应用程序。

日志

App Engine 网络服务器会捕捉处理程序脚本写入标准输出流，以响应网络请求的所有内容。它还会捕捉处理程序脚本写入标准错误流的所有内容，并将其存储为日志数据。您可以使用管理控制台查看和分析您的应用程序的日志数据，或使用 appcfg.py request_logs 下载日志数据。

App Engine Python 运行时环境包括对日志模块的特殊支持，请从 Python 标准库了解日志概念，例如日志级别（[调试]、[信息]、[警告]、[错误]、[严重]）。

```
import logging

from google.appengine.api import users
from google.appengine.ext import db

user = users.get_current_user()
if user:
    q = db.GqlQuery("SELECT * FROM UserPrefs WHERE user = :1", user)
    results = q.fetch(2)
  if len(results) > 1:
        logging.error("more than one UserPrefs object for user %s", str(user))
  if len(results) == 0:
        logging.debug("creating UserPrefs object for user %s", str(user))
        userprefs = UserPrefs(user=user)
        userprefs.put()
  else:
        userprefs = results[0]
else:
    logging.debug("creating dummy UserPrefs for anonymous user")
```

环境

执行环境包含多个对应用程序有用的环境变量。这些环境变量中有一些是 App Engine 特有的，而其他的则是 CGI 标准的一部分。Python 代码可使用 os.environ 参照表访问这些变量。

以下环境变量是 App Engine 特有的：

- APPLICATION_ID：当前运行的应用程序的 ID。
- CURRENT_VERSION_ID：当前运行的应用程序的主要版本和次要版本，表示为 [X.Y]。在应用程序的 app.yaml 文件中指定了主要版本号（[X]）。将应用程序的每个版本上传到 App Engine 时，都会自动设置次要版

本号（[Y]）。在开发网络服务器上，次要版本号一律为 [1]。

- AUTH_DOMAIN：用于通过用户 API 验证用户的域。在 appspot.com 上托管的应用程序具有 gmail.com 的 AUTH_DOMAIN，并且可以接受任何 Google 帐户。通过使用 Google 企业应用套件在自定义域上托管的应用程序具有 AUTH_DOMAIN（相当于该自定义域）。

以下环境变量属于 CGI 标准的一部分，且在 App Engine 中具有特殊的行为：

- SERVER_SOFTWARE：在开发网络服务器中，该值为 [Development/X.Y]，其中 [X.Y] 为此运行时的版本。

其他环境变量则根据 CGI 标准进行设置。有关这些变量的详细信息，请参阅 CGI 标准。

提示：以下 webapp 请求处理程序将在浏览器中显示对应用程序可见的每个环境变量。

```
from google.appengine.ext import webapp
import os

class PrintEnvironmentHandler(webapp.RequestHandler):
  def get(self):
    for name in os.environ.keys():
      self.response.out.write("%s = %s<br />\n" % (name, os.environ[name]))
```

开发网络服务器

App Engine SDK 包含模拟 App Engine Python 运行时环境的网络服务器应用程序。开发服务器：

- 重现模块导入限制，且仅允许处理程序从标准库、包含在 App Engine Python 环境中的第三方库以及应用程序目录中的模块中导入允许的模块
- 重现应用程序缓存行为
- 使用本地文件模拟 App Engine 数据库
- 使用可接受任何电子邮件地址的登录和退出页面模拟 Google 帐户
- 通过从您的计算机中直接抓取网址模拟网址抓取服务
- 使用您选择的 SMTP 服务器或 Sendmail 配置模拟邮件服务

请注意，dev_appserver.py 不模拟配额或限制（例如请求超时）。

请确保使用 Python 2.5 运行 dev_appserver.py。它可在 Python 2.4 环境下运行，但 Python 2.4 和 2.5 之间存在的差别可能会影响您的应用程序（例如处理 Unicode 字符串）。

有关详细信息，请参阅开发网络服务器。

数据库 API

App Engine 数据库用简单的 API 提供查询引擎和事务存储，都在 Google 的可扩展结构上运行。Python 接口包含了数据建模 API 和类似 SQL 的查询语言（称为 GQL），可以让开发可扩展数据库应用程序的工作与进行简单网页

托管一样轻松。

本参考包括以下小节：

概述

App Engine 可扩展数据库在数据对象（称为实体）上存储和执行查询。实体有一个或多个属性（若干个支持的数据类型中某一个类型的命名的值）。属性可以是对其他实体的引用，以创建一对多或多对多关系。

数据库可以在一个事务中执行多种操作，如果操作失败则回滚整个事务。这对于分布式网络应用程序尤其有用，在这种分布式网络应用中多个用户可能同时访问或处理同一数据对象。如果多个用户同时进行更改，数据库会多次重试事务。

与传统数据库不同，该数据库使用分布式架构管理扩展为大型数据集。App Engine 应用程序可以通过描述数据对象之间的关系，以及定义查询的索引，来优化数据的分布方式。

该 App Engine 数据库不是关系数据库。虽然数据库接口有许多与传统数据库相同的功能，但数据库的独特特征采用不同的方式设计和管理数据，以便利用自动扩展功能。

数据库 API 拥有一个用于定义数据模型的机制。Model 描述实体的类型，包括其属性的类型和配置。应用程序使用 Python 类定义 Model，其中 Model 属性描述了属性。某个类型的实体由对应 Model 类的实例表示，其中实例属性则代表属性值。可以通过调用类的构造函数创建实体，然后通过调用 put() 方法进行存储。

```
from google.appengine.ext import db
```

```
from google.appengine.api import users

class Pet(db.Model):
    name= db.StringProperty(required=True)
    type= db.StringProperty(required=True, choices=set(["cat", "dog", "bird"]))
    birthdate= db.DateProperty()
    weight_in_pounds= db.IntegerProperty()
    spayed_or_neutered= db.BooleanProperty()
    owner= db.UserProperty()

pet= Pet(name="Fluffy",
                type="cat",
                owner=users.get_current_user())
pet.weight_in_pounds= 24
pet.put()
```

数据库 API 提供两种用于查询的接口：查询对象接口和类似于 SQL 的查询接口（名为 GQL）。查询以 Model 类的实例的形式返回实体，这些 Model 类可以被修改并放回到数据库中。

```
if users.get_current_user():
    user_pets= db.GqlQuery("SELECT * FROM Pet WHERE pet.owner = :1",
                                    users.get_current_user())
  for petin user_pets:
      pet.spayed_or_neutered= True

    db.put(user_pets)
```

实体和模型

数据库实体有一个键和一组属性。应用程序使用数据库 API 定义数据模型，并创建要存储为实体的模型的实例。模型可为 API 所创建的实体提供通用结构，并可以定义用于验证属性值的规则。

数据库实体

App Engine 数据库中的数据对象称为实体。实体有一个或多个属性（若干个支持的数据类型中某一个类型的命名的值）。

每个实体还有一个唯一标识该实体的键。最简单的键具有数据库提供的类型和唯一的数字 ID。ID 还可以是应用程序提供的字符串。有关键的详细信息，请参阅键和实体组。

应用程序可以通过使用实体的键，或通过执行匹配实体属性的查询，从数据库中抓取实体。查询还可以匹配键中的[祖先]，请参阅键和实体组。查询可以返回零个或多个实体，并可以返回按属性值排序的结果。查询还可以限制数据库返回的结果的数量，以节省内存和运行时间。

与关系数据库不同，App Engine 数据库不要求指定类型的所有实体要有相同的属性。应用程序可以使用模型 API 指定和强制执行其数据模型。

Model 接口

应用程序介绍了它与 Model 配合使用的数据的类型。模型是从 Model 类继承的 Python 类。Model 类定义了数据库实体的新类型，以及该类型将采用的属性。

Model 属性使用 Model 类中的类属性定义。每个类属性都是 Property 类的子类实例，通常是提供的 Property 类之一。Property 实例保留了属性的配置，例如实例是否必须要有该属性才有效，或用于实例的默认值（如果没有提供的话）。

```
from google.appengine.ext import db

class Pet(db.Model):
  name= db.StringProperty(required=True)
  type= db.StringProperty(required=True, choices=set(["cat", "dog", "bird"]))
  birthdate= db.DateProperty()
  weight_in_pounds= db.IntegerProperty()
  spayed_or_neutered= db.BooleanProperty()
  owner= db.UserProperty(required=True)
```

某个定义的实体类型的实体在 API 中用对应的 Model 类的实例表示。应用程序可以通过调用类的构造函数创建新实体。应用程序使用实例的属性来访问和控制实体的属性。Model 实例构造函数接受属性的初始值作为关键字参数。

```
from google.appengine.api import users

pet= Pet(name="Fluffy",
         type="cat",
         owner=users.get_current_user())
pet.weight_in_pounds= 24
```

注意：Model 类的属性是模型属性的配置，其值为 Property 实例。Model 实例的属性是实际属性值，其值为 Property 类接受的类型。

Model 类会使用 Property 实例来验证分配给 Model 实例属性的值。当第一次构建 Model 实例时以及为实例属性分配新值时，都会进行属性值验证。这确保了属性绝不会有无效值。

由于在构建实例时进行验证，配置为必需属性的任何属性都必须在构造函数中进行初始化。在该示例中，name、type 和 owner 都是必需的属性，所以它们的初始值在构造函数中指定。weight_in_pounds 不是模型所必需的，所以它在开始时未被分配值，到后面才分配了值。

使用构造函数创建的模型的实例直至第一次 [放置] 才会存在于数据库中。请参阅创建、获取和删除数据。

注意： 如同所有 Python 类属性一样，模型属性配置是在第一次导入脚本或模块时初始化的。由于 App Engine 缓存是在请求之间导入模块，因此模块配置可能在某个用户的请求期间进行初始化，并在另一个用户的请求期间重复使用。请勿用请求或当前用户特定的数据初始化 模型属性配置（例如默认值）。有关详细信息，请参阅应用程序缓存。

Expando 模型

使用 Model 类定义的模型建立一组固定的属性，该类的每个实例都必须具有这些属性（可能有默认值）。这是一种非常有用的对数据对象进行建模的方法，但是数据库不要求指定类型的每个实体都有相同的属性集。

有时，实体有非必要属性（例如其他同类实体的属性）其实是非常有用的。这样的实体在数据库 API 中由 [Expando] 模型表示。Expando Model 类是 Expando 超类的子类。分配给 Expando 模型实例的属性的任何值都将成为数据库实体的属性，并使用该属性的名称。这些属性称为动态属性。类属性中使用 Property 类实例定义的属性为固定属性 。

Expando 模型可以拥有固定属性和动态属性。 Model 类仅使用固定属性的 Property 配置对象来设置类属性。应用程序为动态属性分配值时将创建动态属性。

```
class Person(db.Expando):
  first_name= db.StringProperty()
  last_name= db.StringProperty()
  hobbies= db.StringListProperty()

p= Person(first_name="Albert", last_name="Johnson")
p.hobbies= ["chess", "travel"]

p.chess_elo_rating= 1350

p.travel_countries_visited= ["Spain", "Italy", "USA", "Brazil"]
p.travel_trip_count= 13
```

由于动态属性没有模型属性定义，所以动态属性未经验证。所有动态属性都可以有任何数据库基本类型的值，包括 None。两个同类实体的相同动态属性可以有不同的值类型，并且可以有一个不设置属性，而另一个设置。

与固定属性不同，动态属性不需要已存在。值为 None 的动态属性与不存在的动态属性不同。如果 Expando 模型实例没有属性的属性，对应的数据实体将没有该属性。您可以通过删除该属性来删除动态属性。

```
del p.chess_elo_rating
```

在过滤条件中使用动态属性的查询将仅返回其属性值与查询中使用的值的类型相同的实体。同样，查询将仅返回设置了该属性的实体。

```
p1= Person()
p1.favorite= 42
p1.put()

p2= Person()
p2.favorite= "blue"
p2.put()

p3= Person()
p3.put()

people= db.GqlQuery("SELECT * FROM Person WHERE favorite < :1", 50)
# people has p1, but not p2 or p3

people= db.GqlQuery("SELECT * FROM Person WHERE favorite > :1", 50)
# people has no results
```

Expando 类是 Model 类的子类，并且继承其所有方法。

属性和类型

数据库支持一组固定的实体属性的值类型，包括 Unicode 字符串、整数、浮点数、日期、实体键、字节字符串（二进制大对象）和各种 GData 类型。每个数据库值类型都有 google.appengine.ext.db 模块提供的对应的 Property 类。

类型和 Property 类介绍了所有支持的值类型及其对应的 Property 类。下面介绍几个特殊的值类型。

字符串、长字符串和二进制大对象

数据库支持存储文本的两种值类型：不到 500 个字节的短文本字符串和 500 个字节以上的长文本字符串。短字符串会编入索引并可在查询过滤条件和排序顺序中使用。长字符串不会编入索引且不能在过滤条件或排序顺序中使用。

短字符串值可以是 unicode 值或 str 值。如果值是 str，则假设编码为 'ascii'。要为 str 值指定一种不同的编码，可以用 unicode() 类型的构造函数将其转换为 unicode 值，这种类型的构造函数会使用 str 和编码名称作为参数。可以使用 StringProperty 类对短字符串进行建模。

```
class MyModel(db.Model):
  string = db.StringProperty()

obj= MyModel()
```

```
# Python Unicode literal syntax fully describes characters in a text string.
obj.string = u"kittens"

# unicode() converts a byte string to a Unicode value using the named codec.
obj.string = unicode("kittens", "latin-1")

# A byte string is assumed to be text encoded as ASCII (the 'ascii' codec).
obj.string = "kittens"

# Short string properties can be used in query filters.
results= db.GqlQuery("SELECT * FROM MyModel WHERE string = :1", u"kittens")
```

长字符串值由 db.Text 实例表示。其构造函数采用 unicode 值或 str 值，并可能会使用 str 中使用的编码名称。可以使用 TextProperty 类对长字符串进行建模。

```
class MyModel(db.Model):
  text= db.TextProperty()

obj= MyModel()

# Text() can take a Unicode value.
obj.text= db.Text(u"lots of kittens")

# Text() can take a byte string and the name of an encoding.
obj.text= db.Text("lots of kittens", "latin-1")

# If no encoding is specified, a byte string is assumed to be ASCII text.
obj.text= db.Text("lots of kittens")

# Text properties can store large values.
obj.text= db.Text(open("a_tale_of_two_cities.txt").read(), "utf-8")
```

数据库还支持非文本字节字符串（或 [二进制大对象]）类型。和长文本字符串一样，二进制大对象不会编入索引，也不能在查询过滤条件或排序顺序中使用。Blob 实例代表字节字符串，并使用 str 值作为其构造函数的参数。使用 BlobProperty 类对二进制大对象进行建模。

```
class MyModel(db.Model):
  blob= db.BlobProperty()

obj= MyModel()

obj.blob= db.Blob(open("image.png").read())
```

列表

属性可以有多个值，在数据库 API 中表示为 Python list。列表可以包含数据库支持的任何值类型的值。一个列表属性甚至可以有不同类型的值。顺序将保留，因此当查询和 get() 返回实体时，列表属性值的顺序将与存储这些值时的顺序相同。

ListProperty 类对列表进行建模，并强制列表中的所有值都采用指定类型。为了方便起见，库还提供 StringListProperty，类似于 ListProperty(basestring)。

```
class MyModel(db.Model):
  numbers= db.ListProperty(long)

obj= MyModel()
obj.numbers= [2, 4, 6, 8, 10]

obj.numbers= ["hello"]   # ERROR: MyModel.numbers must be a list of longs.
```

列表属性上的查询过滤将针对列表的成员测试指定的值。如果至少一个列表成员符合条件，则条件为 True。

```
# Get all entities where numbers contains a 6.
results= db.GqlQuery("SELECT * FROM MyModel WHERE numbers = 6")

# Get all entities where numbers contains at least one element less than 10.
results= db.GqlQuery("SELECT * FROM MyModel WHERE numbers < 10")
```

仅对列表成员进行查询过滤。无法在一个查询过滤中测试两个列表的相似之处。

数据库在内部将列表属性值表示为该属性的多个值。如果列表属性值为空列表，则该属性在数据库中没有表示。对于静态属性（具有 ListProperty）和动态属性，数据库 API 处理这种情况的方式不同：

- 可以为静态 ListProperty 分配空列表作为值。该属性不存在于数据库中，但是 Model 实例的行为就像值是空列表一样。静态 ListProperty 的值不能为 None。
- 无法为具有 list 值的动态属性分配空列表值。但是，它可以有 None 值，并可以被删除（使用 del）。

ListProperty 模型测试添加到列表的值的类型是否正确，如果不正确，则抛出 BadValueError。即使是在检索以前存储的实体并将其加载到模型中时，也会进行该测试（并可能失败）。由于 str 值在存储前转换为 unicode 值（作为 ASCII 文本），ListProperty(str) 会被视为 ListProperty(basestring)，即接受 str 和 unicode 值的 Python 数据类型。您还可以使用 StringListProperty() 实现此目的。

要存储非文本字节字符串，请使用 db.Blob 值。当存储和检索二进制大对象字符串的字节时，将保留这些字节。您可以将形式为二进制大对象列表的属性声明为 ListProperty(db.Blob)。

列表属性以独特的方式与排序顺序交互。有关详细信息，请参阅排序顺序和列表属性。

引用

一个属性值可以包含另一个实体的键。该值为 Key 实例。

ReferenceProperty 类对键值进行建模，并强制所有值参考指定类型的实体。为了方便起见，库还提供 SelfReferenceProperty，等同于参考具有该属性的实体的同一类型的实体的 ReferenceProperty。

向 ReferenceProperty 属性分配 Model 实例将自动使用其键作为值。

```
class FirstModel(db.Model):
  prop= db.IntegerProperty()

class SecondModel(db.Model):
  reference= db.ReferenceProperty(FirstModel)

obj1= FirstModel()
obj1.prop= 42
obj1.put()

obj2= SecondModel()

# A reference value is the key of another entity.
obj2.reference= obj1.key()

# Assigning a model instance to a property uses the entity's key as the value.
obj2.reference= obj1
obj2.put()
```

可以将 ReferenceProperty 属性值当作是引用的实体的 Model 实例使用。如果引用的实体不在内存中，则当使用该属性作为实例时将会自动从数据库中抓取实体。

```
obj2.reference.prop= 999
obj2.reference.put()

results= db.GqlQuery("SELECT * FROM SecondModel")
another_obj= results.fetch(1)[0]
v= another_obj.reference.prop
```

当删除键是引用属性的值的实体时，引用属性不会发生更改。引用属性值可以是不再有效的键。如果某个应用程序预计引用可能无效，则它可以使用 if 语句测试对象是否存在：

```
obj1= obj2.reference

if not obj1:
 # Referenced entity was deleted.
```

ReferenceProperty 有另一个便利功能：反向引用。如果一个模型将 ReferenceProperty 传递给另一个模型时，每个引用的实体都会获得一个属性，该属性的值为返回第一个引用它的模型的所有实体的 Query。

```
# To fetch and iterate over every SecondModel entity that refers to the
# FirstModel instance obj1:
for objin obj1.secondmodel_set:
  # ...
```

反向引用属性的名称默认为 modelname_set（使用以小写字母表示的 Model 类的名称，并在末尾添加 [_set]），并可以使用 collection_name 参数调整为 ReferenceProperty 构造函数。

如果您有多个参照同一 Model 类的 ReferenceProperty 值，反向引用属性的默认结构将抛出错误。

```
class FirstModel(db.Model):
   prop= db.IntegerProperty()

# This class raises a DuplicatePropertyError with the message
# "Class Firstmodel already has property secondmodel_set"
class SecondModel(db.Model):
   reference_one= db.ReferenceProperty(FirstModel)
   reference_two= db.ReferenceProperty(FirstModel)
```

要避免这种错误，您必需明确地设置 collection_name 参数：

```
class FirstModel(db.Model):
   prop= db.IntegerProperty()

# This class runs fine
class SecondModel(db.Model):
   reference_one= db.ReferenceProperty(FirstModel,
         collection_name="secondmodel_reference_one_set")
   reference_two= db.ReferenceProperty(FirstModel,
         collection_name="secondmodel_reference_two_set")
```

Model 实例的自动引用和取消引用、类型检查和反向引用仅在使用 ReferenceProperty 模型属性类时可用。存储为 Expando 动态属性的值或 ListProperty 值的键没有这些功能。

属性名称

数据库保留所有以两个下划线字符 (__*__) 开头和结尾的属性名称。应用程序无法创建具有这样的名称的属性。

在 Python API 中，将忽略名称以下划线 (_) 开头的 Model 实例的属性，且不保存到数据库实体。这允许您在 Model 实例上存储值以供临时内部使用，而不影响与实体一起保存的数据。

由于默认情况下 Python API 使用 Model 实例的属性作为属性名称，因此，已被实例模型使用的属性不可直接作为属性的属性名称使用。同样，没有被 Model 构造函数的关键字参数使用的名称可作为属性名称使用。请参阅保留的属性名称列表。

数据库本身允许使用这些名称。如果应用程序需要数据库实体有一个名称类似于 Python API 中保留的字词的属性，应用程序可以使用固定属性，并将 name 参数传递到 Property 类构造函数。请参阅 Property 类构造函数。

```
class MyModel(db.Model):
  obj_key= db.StringProperty(name="key")
```

创建、获取和删除数据

数据库 API 代表作为 Model 类实例的实体。Model 实例创建、更新和删除实体的方法。可以使用 Query 或 Key，以 Model 实例的形式从数据库抓取实体。

创建和更新实体

Model（和 Expando）类的实例代表数据库实体。应用程序通过调用对应类别的 Model 类的构造函数来创建指定类别的新实体。

```
pet= Pet(name="Fluffy",
         type="cat",
         owner=users.get_current_user())
```

新实体不会在数据库中创建，直至第一次 [放置] 该实例，方式是通过对实例调用 put() 方法，或通过将实例传递到 db.put() 函数。

```
pet.put()
```

```
db.put(pet)
```

如果之前已经存储了实例，则 put() 方法会更新现有实体。

查询以 Model 实例的形式返回结果。这些实体可以进行修改并放回到数据库中。

```
if users.get_current_user():
  user_pets= db.GqlQuery("SELECT * FROM Pet WHERE pet.owner = :1",
                         users.get_current_user())
 for petin user_pets:
    pet.spayed_or_neutered= True

  db.put(user_pets)
```

使用 Query 获取实体

数据库可用在指定类型的实体之间执行查询。查询可以使用实体属性值必须满足的条件过滤结果，并可以返回按属性值排序的结果。查询还可以用指定的祖先限制实体的范围，请参阅键和实体组。

有关查询的工作方式的完整说明，包括一些查询无法执行的操作，请参阅查询和索引。

数据库 API 提供两种接口用于对实体属性执行查询：Query，这是一种使用 Query 对象上的方法来准备查询的接口；GqlQuery，这是一种使用名为 GQL 的查询语言（类似于 SQL）的接口。

Query 接口

Model（或 Expando）类上的 all() 方法返回代表对应类别的所有实体查询的 Query 对象。应用程序通过在对象上调用 filter()、order() 和 ancestor() 方法来准备查询。

```
class Story(db.Model):
  title= db.StringProperty()
  date= db.DateTimeProperty()

query= Story.all()

query.filter('title =', 'Foo')
query.order('-date')
query.ancestor(key)

# These methods can be chained together on one line.
query.filter('title =', 'Foo').order('-date').ancestor(key)
```

GqlQuery 接口

GqlQuery 类构造函数使用 GQL 查询字符串和可选的参数进行绑定。查询字符串指定类型、过滤器、排序顺序和祖先条件。查询字符串还可以包含结果限制和偏移。

```
# Parameters can be bound with positional arguments.
query= db.GqlQuery("SELECT * FROM Story WHERE title = :1 "
                   "AND ANCESTOR IS :2 "
                   "ORDER BY date DESC",
                   'Foo', key)

# Or, parameters can be bound with keyword arguments.
query= db.GqlQuery("SELECT * FROM Story WHERE title = :title "
                   "AND ANCESTOR IS :parent "
                   "ORDER BY date DESC",
                    title='Foo', parent=key)
```

```
# String, number and Boolean values can be literal values in the string.
query= db.GqlQuery("SELECT * FROM Story WHERE title = 'Foo' "
                   "AND ANCESTOR IS :parent "
                   "ORDER BY date DESC",
                    parent=key)
```

Model 类的 gql() 类方法还可以准备来自字符串的 GqlQuery 对象。该字符串是忽略了 SELECT * FROM Model 的 GQL 查询字符串，因为该部分是暗含的。

```
query= Story.gql("WHERE title = :title "
                 "AND ANCESTOR IS :parent "
                 "ORDER BY date DESC",
                  title='Foo', parent=key)
```

参数绑定可以使用 bind() 方法重新绑定到新值。应用程序可以通过重新绑定参数和重新执行查询来重复使用 GqlQuery 对象。

执行查询和访问结果

Query 和 GqlQuery 对象在应用程序尝试访问结果之前不会执行查询。当应用程序访问结果时会执行查询，且结果会作为查询的 Model 类的实例加载到内存中。两种 Query 类都提供两种执行查询和访问结果的方式：fetch() 方法和迭代器接口。

fetch() 方法会使用最大数目的结果来抓取（限制），并会跳过可选数目的结果（偏移）。该方法会执行查询，然后抓取结果，直至达到限制或没有更多的结果为止。一旦将 结果加载到内存中，它将跳过偏移（如果指定了偏移），然后以 Model 实例的列表形式返回请求的结果。将针对每个对 fetch() 的调用执行完全查询。

注意：偏移不影响从数据库抓取的结果的数量。将会抓取所有结果直至达到限制为止，并将其存储在内存中。偏移仅影响 fetch() 方法返回的内容。

```
results= query.fetch(10)
for result in results:
  print "Title: " + result.title
```

指定给 fetch() 方法的限制和偏移覆盖任何在 GQL 查询字符串中指定的限制和偏移。

如果 Query 对象用作迭代器，则查询将在没有限制或偏移的情况下执行，结果会加载到内存中，且返回的值是结果上的迭代器。迭代器产生 Model 类的实例。

```
for result in query:
  print "Title: " + result.title
```

注意：数据库最多能返回 1000 个结果来响应查询，与用于抓取结果的限制和偏移无关。1000 个结果包含了任何使用偏移跳过的结果，所以如果结果超过 1000 个的查询使用 100 作为偏移，则会返回 900 个结果。

使用 Key 获取实体

将实体存储在数据库中后，该实体将具有唯一的键。Key 值在 API 中表达为 Key 类的实例。Model 实例的 put() 方法和 db.put() 函数返回存储的实体的 Key。首次存储 Model 实例后，Model 实例的 key() 方法将返回实例的 Key。

```
entity.put()
key= entity.key()

# ...

entity= db.get(key)
```

Key 值的常见用法是将其存储为另一个实体上的属性的值。ReferenceProperty Model Property 类提供对 Model 实例（以键的形式）的自动引用和取消引用：Model 实例可以直接分配给 ReferenceProperty，并且它的键会被用作值。

```
class Pet(db.Model):
  name= StringProperty()
  owner= ReferenceProperty(PetOwner)

class PetOwner(db.Model):
  name= StringProperty()

owner= PetOwner(name="Albert")
pet= Pet(name="Fluffy", owner=owner)

# This is equivalent:
pet= Pet(name="Fluffy", owner=owner.key())
```

同样，通过属性访问的 ReferenceProperty 值的行为类似其实例。数据实体自动抓取，并且在使用时才抓取。

```
pets= GqlQuery("SELECT * FROM Pet WHERE name = :1", "Fluffy")
pet= pets.get()

owner_name= pet.owner.name
```

不使用 ReferenceProperty 模型（例如采用 Expando 动态属性或 ListProperty 元素）存储的键值没有自动取消引用的行为。

db.get() 函数从数据库抓取实例以获得一个 Key（或 Key 的列表）。

Key 可以编码为字符串，以便在应用程序外部传递。要将编码为字符串的键转换回 Key 对象，应用程序会将该字符串传递到 Key 构造函数。

```
obj= MyModel(name="Foo")
```

```
self.response.write('<a href="/view?key=%s">%s</a>' % (str(obj.key()),
                                                       obj.name()))

# ...

key_name= self.request.get('key')
obj= db.get(db.Key(key_name))
```

注意： Key 的字符串编码是不透明的，但不进行加密。如果您的应用程序要使键无法猜测，您应在将其发送给用户之前进一步加密字符串编码的 Key。

删除实体

应用程序可以使用 Model 实例或 Key 从数据库中删除实体。Model 实例的 delete() 方法会从数据库中删除对应的实体。delete() 函数采用 Key 或 Key 的列表并从数据库中删除实体。

```
q= db.GqlQuery("SELECT * FROM Message WHERE create_date < :1", earliest_date)
results= q.fetch(10)
for result in results:
  result.delete()

# or...

q= db.GqlQuery("SELECT * FROM Message WHERE create_date < :1", earliest_date)
results= q.fetch(10)
db.delete(results)
```

删除实体不会更改数据库中可能参考该实体的 Key 值。如果您的应用程序可以尝试取消引用删除的实体的 Key 值，则该应用程序应使用 db.get() 执行该操作，然后在访问属性前测试返回值。

删除作为其他实体的祖先的实体不会影响这些子实体。只要应用程序不依赖于祖先的存在构建子孙实体的键，应用程序将仍能够访问子孙实体。

键和实体组

数据库中的每个实体都有一个键，一个对于应用程序中所有实体都唯一的标识符。一个键有若干个组成部分：路径描述两个实体之间的父子关系，实体的类型，以及由应用程序分配给实体的名称，或由数据库分配的数字 ID。

- 类型、名称和 ID
- 实体组、祖先和路径
- 路径和键唯一性

类型、名称和 ID

每个实体都属于特定的类型，类型是可以由查询返回的一组实体。与表中的行不同，虽然应用程序可以在一个数据模型中建立此类限制，然而相同类型的两个实体不需要有相同的属性。数据库 API 使用 Model（或 Expando）子类的名称作为类型的名称。

例如，该类定义了名为 [Story] 的类型的 Model。

```
class Story(db.Model):
  title= db.StringProperty()
  author= db.StringProperty()
```

每个实体都有标识符。应用程序可以通过授予实例构造函数 key_name 参数（str 值）来分配自己的标识符以便在键中使用。

```
s= Story(key_name="xzy123")
```

key_name 存储为 Unicode 字符串，str 值转换为 ASCII 文本。key_name 绝不能以数字开头，绝不能采用 __*__ 形式（以两根下划线开头和结尾）。如果您的应用程序使用用户提交的数据作为实体键名（例如电子邮件地址），应用程序应首先清理值，例如用已知的字符串作为前缀以符合这些要求。

如果未指定 key_name，则第一次将实体存储到数据库中时，会向其分配数字 ID。

```
s2= Story()      # s2 does not have a name or an ID.
s2.put()         # s2 is given an ID by the datastore.
```

一旦创建了实体，就不能更改其 ID 或名称。

提示：不能用与查询中的属性值类似的方式使用键名和 ID。但是，您可以使用命名的键，然后将该名称作为属性存储。您可以通过存储对象以分配 ID、使用 obj.key().id() 获取 ID 值、用 ID 设置属性，然后再次存储对象，以对数字 ID 执行类似的操作。

实体组、祖先和路径

每个实体都属于一个实体组，它是可以在一个事务中控制的一组实体（一个或多个）。实体组关系会让 App Engine 在分布式网络的相同部分中存储若干实体。事务会针对实体组设置数据库操作，且所有操作都会以组的形式应用。如果事务失败，则全都不应用。

当应用程序创建一个实体时，它将另一个实体分配为新实体的父实体。向新实体分配父实体会将新实体放置在与父实体相同的实体组。

没有父实体的实体是根实体。作为另一个实体的父实体的实体也可以有父实体。从某实体到根的父实体链是该实体的路径，路径的成员是该实体的祖先。实体的父实体是在创建该实体时定义的，且以后不能再更改。

每个采用指定根实体作为祖先的实体都在相同的实体组中。一个组中的所有实体都存储在相同的数据库节点中。一个事务可以修改一个组中的多个实体，或向组添加新实体（方法是以组中的现有实体作为新实体的父实体）。

有关事务的详细信息，请参阅事务。

如果删除了某个作为其他实体的祖先的实体，则后续实体不会被删除。仍可以使用后续实体完整的键或路径对其进行访问。

您可以用祖先路径创建实体而无需先创建父实体。为此，您可以使用类型和键名为祖先创建一个 Key，然后将其用作新实体的父实体。所有具有同一根祖先的实体都属于同一实体组，与路径的根是否代表实际实体无关。

使用实体组的提示：

- 仅当事务需要实体组时才使用实体组。对于实体之间的其他关系，请使用可以在查询中使用的 ReferenceProperty 属性和 Key 值。
- 您 的应用程序拥有的实体组越多（即有更多根实体），数据库就能越有效地在数据库节点之间分布实体组。更好的分布可提高创建和更新数据的性能。此外，多个用户 尝试同时更新相同实体组中的实体会导致部分用户重新尝试他们的事务，因而可能导致某些用户无法提交更改。请勿将应用程序的所有实体放在一个根下。
- 较好的实体组做法是：它们应与单个用户的数据值的大小相当或更小。
- 实体组对查询的速度没有明显影响。

路径和键唯一性

实体的完整键（包括路径、类型、名称或数字 ID）都是唯一的并特定于该实体。在数据库中创建实体时，会分配完整的键，且不能更改其任何部分。

只要至少一个部分不同，两个不同实体的键就可以有相似的部分。例如，如果两个实体有不同的父实体，它们可以有相同的类型和名称。类似地，如果两个实体的类型不同，它们可以有相同的父实体（或没有父实体）和名称。

应用程序不应依赖于以增序（实体创建的顺序）分配的数字 ID。这是通常情况，但并无保证。

查询和索引

每个数据库查询使用一个索引，即一个包含按照所需顺序排列的查询结果的表格。App Engine 应用程序会在一个名为 index.yaml 的配置文件中定义其索引。开发网络服务器在遇到未配置索引的查询时会自动为该文件添加建议。您可以通过在上传该应用程序之前编辑该文件来手动调整索引。

基于索引的查询机制支持大多数常见查询类型，但不支持您可能惯用的来自其他数据库技术的一些查询。以下描述了对查询的限制及其对此所做的说明。

- 引入查询
- 引入索引
- 用 index.yaml 定义索引
- 对查询的限制
- 大的实体和分解索引

引入查询

查询从数据库中检索满足一组条件的实体。查询会指定一种实体、基于属性值的零个或多个条件（有时称作 [过滤器]）以及零个或多个排序顺序描述。执行查询时，它会抓取指定类型中满足所有指定条件并按照描述的顺序排序的全部实体。

数据库 API 提供两种接口来准备和执行查询：Query 接口（使用不同方法准备查询）和 GqlQuery 接口（使用一种称为 GQL 的类似 SQL 的查询语言从查询字符串准备查询）。在创建、获取和删除数据：使用 Query 获取实体以及相应的参考页面中更详细地介绍了这些接口。

```
class Person(db.Model):
    first_name= db.StringProperty()
    last_name= db.StringProperty()
    city= db.StringProperty()
    birth_year= db.IntegerProperty()
    height= db.IntegerProperty()

# The Query interface prepares a query using instance methods.
q= Person.all()
q.filter("last_name =", "Smith")
q.filter("height <", 72)
q.order("-height")

# The GqlQuery interface prepares a query using a GQL query string.
q= db.GqlQuery("SELECT * FROM Person " +
                       "WHERE last_name = :1 AND height < :2 " +
                       "ORDER BY height DESC",
                       "Smith", 72)

# The query is not executed until results are accessed.
results= q.fetch(5)
for p in results:
  print "%s %s, %d inches tall" % (p.first_name, p.last_name, p.height)
```

引入索引

App Engine 数据库会为应用程序要进行的每个查询都保留一个索引。当应用程序对数据库实体做出更改时，数据库会使用正确的结果更新索引。当应用程序执行查询时，数据库会直接从相应的索引中抓取结果。

应用程序对查询中使用的每个类型、过滤器属性和操作符以及排序顺序的组合都具有一个索引。请考虑上述示例查询：

```
SELECT * FROM Person WHERE last_name = "Smith"
```

```
                AND height < 72

            ORDER BY height DESC
```

该查询的索引是 Person 类型实体的键表，其中包括 height 和 last_name 属性的值列。该索引按照 height 的降序排序。

形式相同但过滤器值不同的两个查询会使用相同的索引。例如，下面的查询与上面的查询使用相同的索引：

```
SELECT * FROM Person WHERE last_name = "Jones"
                AND height < 63

            ORDER BY height DESC
```

数据库按照以下步骤执行查询：

1. 数据库会标识符合查询的种类、过滤器属性、过滤器操作符和排序顺序的索引。
2. 数据库会使用该查询的过滤器值在满足全部过滤器条件的第一个实体处开始扫描该索引。
3. 数据库会继续扫描该索引并返回每个实体，直到发现下一个不满足过滤器条件的实体或到达到该索引末尾。

索引表包含在过滤器或排序顺序中使用的每个属性列。行的排序会按照以下几个方面进行：

- 祖先
- 在等式或 IN 过滤器中使用的属性值
- 在不等式过滤器中使用的属性值
- 在排序顺序中使用的属性值

注意：要发挥索引的作用，可像处理 = 过滤器一样处理 IN 过滤器，像处理其他不等式过滤器一样处理 != 过滤器。

这可将使用该索引的每项可能查询的所有结果以连续行的形式排在表格中。

该机制可支持许多查询，且适用于大部分应用程序。然而，该机制不支持您可能惯用的来自其他数据库技术的一些类型的查询。请参阅下面的对查询的限制。

提示：查询过滤器没有一种明确的方法来与部分字符串值相匹配，然而您可以使用不等式过滤器

```
db.GqlQuery("SELECT * FROM MyModel WHERE prop >= :1 AND prop < :2", "abc", u"abc" + u"\xEF\xBF\xBD")
```

仿造一个前缀匹配。这样可以匹配字符串属性 prop 以 abc 字符开头的每个 MyModel 实体。字节字符串 "\xEF\xBF\xBD" 表示可能存在的最大 Unicode 字符。当属性值在索引中进行排序时，属于此范围的值是以指定的

前缀开头的所有值。

Query 永远不会返回没有已过滤属性的实体

索引仅包含其每个属性都由该索引引用的实体。如果实体没有由某索引引用的属性，那么该实体将不会显示在该索引中，且永远不会成为使用该索引的查询的结果。

请注意，App Engine 数据库区分不具有属性的实体和具有带空值（在 Python 中为 None）的属性的实体。如果您希望一类实体中的每个实体都成为可通过查询可搜索到的结果，则可以使用数据模型将一个默认值（例如 None）分配到过滤器在查询中所使用的属性。

Text 和 Blob 值未编入索引

具有 Text 或 Blob（例如具有 TextProperty 或 BlobProperty 模型）类型的值的属性不包含在索引中，因此，无法通过查询找到这些属性。要对短字符串值进行过滤，请使用字符串或 Unicode 值（StringProperty 模型）。

不将这些属性值编入索引的结果是，属性上带有过滤器或排序顺序的查询将永远不会匹配属性值为 Text 或 Blob 的实体。带有此类值的属性会表现得如同未对该属性的查询过滤器和排序顺序进行设置。

用 index.yaml 定义索引

App Engine 在默认情况下为许多简单查询创建索引。对于其他查询，应用程序必须在一个名为 index.yaml 的配置文件中指定所需的索引。如果在 App Engine 下运行的应用程序试图执行一个无相应索引（默认情况下提供或在 index.yaml 中描述）的查询，该查询会失败。

App Engine 会为以下形式的查询提供自动索引：

- 只使用等式、IN 和祖先过滤器的查询
- 只使用不等式过滤器（只能属于单个属性）的查询
- 只有一种排序顺序（升序）的查询

其他形式的查询需要在 index.yaml 中指定其索引，这类查询包括：

- 按降序排列的查询
- 有多种排序顺序的查询
- 在一个属性上具有一个或多个不等式过滤器，以及在其他属性上具有一个或多个等式或 IN 过滤器的查询
- 具有不等式过滤器和祖先过滤器的查询

开发网络服务器 (dev_appserver.py) 可以让您易于管理 index.yaml：以前无法执行没有索引配置且需要进行索引配置的查询，而开发网络服务器可向文件添加一个索引定义，这样就可执行这种查询了。

如果对您的应用程序进行的本地测试调用该应用程序可能执行的每个查询（祖先、过滤器和排序顺序的每种组合），则生成的条目会表示一组完整的索引。如果您的测试不会应用每种可能的查询形式，您可以在上传该应用程序之前审阅和调整文件中的索引定义。

提示： 如果要以 --require_indexes 选项启动 dev_appserver.py，则会禁止生成 index.yaml，并且需要不存在的索引配置的查询会抛出错误。请使用此选项测试您的应用程序以验证是否所有必需的索引配置都存在。

index.yaml 描述了每个索引表，包括类型、查询过滤器和排序顺序所需的属性以及该查询是否使用了祖先子句（Query.ancestor() 子句或 GQL ANCESTOR IS 子句）。属性会按照将被排序的顺序列出：首先是在等式或 IN 过滤器中使用的属性，接着是在不等式过滤器中使用的属性，最后是查询结果排序顺序及其方向。

请再次考虑以下示例查询：

```
SELECT * FROM Person WHERE last_name = "Smith"
                       AND height < 72
                     ORDER BY height DESC
```

如果应用程序仅执行此查询（也可能是类似此查询但具有 "Smith" 和 72 不同的值的其他查询），那么，index.yaml 文件会以以下形式显示：

```
indexes:

- kind: Person
  properties:
  - name: last_name
  - name: height
    direction: desc
```

当创建或更新了一个实体时，每个相应的索引也进行更新。应用到实体的索引数量会影响创建或更新该实体所需要的时间。

有关 index.yaml 语法的详细信息，请参阅配置索引。

对查询的限制

索引查询机制的本质是对查询功能强加一些限制。

对一个属性进行过滤或排序需要确认该属性确实存在

属性的查询过滤条件或排序顺序也暗含了一个条件，即实体必须具有该属性的值。

数据库实体不需具有其他同类实体所具有的属性值。属性上的过滤器只能与具有该属性的值的实体相匹配。过滤器或排序顺序中所使用的不具有属性值的实体会从为该查询创建的索引中删除。

没有可与不具有属性的实体相匹配的过滤器

无法为缺少指定属性的实体执行查询。一种解决方法是创建一个固定的（已建模的）属性，默认值为 None，然后为实体创建一个过滤器，属性值为 None。

只允许在一个属性上使用不等式过滤器

查询在其所有过滤器中只能在一个属性上使用不等式过滤器（<、<=、>=、> 和 !=）。

例如，允许使用此 GQL 查询：

```
SELECT * FROM Person WHERE birth_year >= :min
                       AND birth_year <= :max
```

然而，不允许使用此 GQL 查询，因为此 GQL 查询在同一个查询中的两个不同的属性上使用了不等式过滤器：

```
SELECT * FROM Person WHERE birth_year >= :min_year
                       AND height >= :min_height        # ERROR
```

过滤器可合并同一查询（包括属性上带有一个或多个不等式条件的查询）中不同属性的同级 (=) 比较。允许执行此操作：

```
SELECT * FROM Person WHERE last_name = :last_name
                       AND city = :city
                       AND birth_year >= :min_year
```

查询机制基于将查询的所有结果彼此相邻地排列在索引表中，以避免为查找结果而不得不扫描整个整个表。在保持所有结果连续出现在表中的同时，单个索引表无法表示用于多个属性上的多个不等式过滤器。

必须在采用其他排序顺序之前对不等式过滤器中的属性进行排序

如果查询具有带不等式比较以及带一个或多个排序顺序的两种过滤器，那么，该查询必须包含在该不等式中使用的属性的排序顺序，且该排序顺序必须出现在其他属性上的排序顺序之前。

此 GQL 查询无效，因为此 GQL 查询使用了不等式过滤器，且未根据已过滤的属性进行排序：

```
SELECT * FROM Person WHERE birth_year >= :min_year
                       ORDER BY last_name                    # ERROR
```

同样，此 GQL 查询无效，因为它在根据其他属性排序之前未根据已过滤的属性进行排序：

```
SELECT * FROM Person WHERE birth_year >= :min_year
                       ORDER BY last_name, birth_year    # ERROR
```

此 GQL 查询有效：

```
SELECT * FROM Person WHERE birth_year >= :min_year
                       ORDER BY birth_year, last_name
```

要获取匹配不等式过滤器的所有结果，查询会扫描该索引表寻找第一个匹配的行，然后返回所有连续的结果，直到查询找到不匹配的行为止。对于表示完整结果集的连续的行，这些行必须先按照不等式过滤器排序，然后再按其他排序顺序排序。

排序顺序和列表属性

由于为列表属性建立索引的方法的缘故，列表值的排序顺序异常：

- 如果实体按照列表属性以升序排列，则进行排序所使用的值会成为该列表中最小的元素。
- 如果实体按照列表属性以降序排列，则进行排序所使用的值会成为该列表中最大的元素。
- 该列表中的其他元素不影响排序顺序，该列表的长度也不会影响排序顺序。
- 就链而言，该实体的键被用作该链的断路器。

此排序顺序引起异常结果：以升序和降序排序时 [1, 9] 都显示在 [4, 5, 6, 7] 之前。

需要特别注意列表属性上的既具有等式过滤器又具有排序顺序的查询。在那些查询中，排序顺序被忽略。对于非列表属性，这仅是一个简单的优化。对于属性来说，每个结果都具有相同的值，因此，不需要对结果进一步排序。

然而，列表属性可能具有其他的值。由于忽略了排序顺序，与应用了排序顺序时相比，可能会以一个不同于前者顺

序的顺序返回查询结果。(恢复已放弃的排序顺序会花费巨大并需要额外的标记，又因为这种使用情况很少出现，因此查询计划员禁止此项操作。)

大的实体和分解索引

如上所述，会把每个实体的每个属性（不具有 Text 或 Blob 值）添加到至少一个索引表（包含默认情况下提供的简单索引和在有关该属性的应用程序的 index.yaml 文件中所描述的任何索引）中。对于每个属性都有一个值的实体，App Engine 在其简单索引中都存储一次属性值，且每次在自定义索引中引用该属性时都存储一次属性值。每次更改该属性的值时，这些索引条目中的每一个都必须进行更新，因此，引用该属性的索引越多，put() 成功更新该属性所需要的时间就越长。

为防止更新实体花费过多时间，数据库会限制单个实体可具有的索引条目的数量。该限制涉及的范围广大，且使用大部分应用程序时将不会引起用户的注意。然而，有些情况下您可能会遇到该限制。例如，具有许多单一值属性的实体可能会超出索引条目的限制。

对于具有多个值的属性，例如使用列表值或 ListProperty 模型，则将每个值作为索引中单独的条目存储。当单个属性具有许多值（很长的一个列表）时，具有该单个属性的实体可以超出索引条目的限制。

引用带有多个值的多个属性的自定义索引可以仅带有几个值，但容量却很大。要完整记录这类属性，索引表必须包含该索引每个属性的值的每一项排列行。例如，以下索引（用 index.yaml 语法描述）包括类型为 MyModel 的实体的 x 和 y 属性：

```
indexes:
- kind: MyModel
  properties:
  - name: x
  - name: y
```

以下代码会创建分别具有属性 x 和属性 y 的 2 个值的实体：

```
class MyModel(db.Expando):
  pass

e2= MyModel()
e2.x= ['red', 'blue']
e2.y= [1, 2]
e2.put()
```

为了正确表示这些值，该索引必须存储 8 个属性值：x 和 y 上的每个内置索引各 2 个，自定义索引中 x 和 y 的每项排列各 1 个。如果存在许多列表值，这可能意味着一个索引必须存储单个实体的许多索引条目。您可以将引用带有多个值的多个属性的索引称作 [分解索引]，因为该类索引可以仅带有几个值，但容量却很大。

如果 put() 会导致产生大量超出限制的索引条目，那么调用会失败，并抛出 BadRequestError 异常。如果您创建一个包含大量索引条目的新索引，且这些索引条目超出了对所创建的任何实体的限制，那么针对该索引的查询会失败，

并且该索引会以 [错误] 的状态显示在管理控制台中。

要处理 [错误] 索引，首先将这些索引从您的 index.yaml 文件中删除，并运行 appcfg.py vacuum_indexes。然后，重新制定索引定义以及相应的查询，或者删除导致该索引 [分解] 的实体。最后，将该索引添加回至 index.yaml 并运行 appcfg.py update_indexes。

您可以通过避免需要自定义索引的查询使用列表属性来避免分解索引。如上所述，这包括以降序排列的查询、多个排序顺序、等式和不等式过滤器的混合使用以及祖先过滤器。

事务

App Engine 数据库支持*事务*。事务是一项操作或一系列操作，要么全部成功，要么全部失败。通过使用 Python 函数对象以及 db.run_in_transaction() 函数，应用程序可在单个事务中执行多项操作。

使用事务

*事务*是一项数据库操作或一系列数据库操作，要么全部成功，要么全部失败。如果事务成功完成，则会对数据库产生所有预期的作用。如果事务失败，则不会起任何作用。

每项数据库写入操作都是不可再分割的。要么执行 put() 或 delete()，要么不执行。如果有太多用户试图同时修改一个实体，那么这种高占用率将可能引发操作失败。当应用程序达到配额限制时，也可能会引发操作失败。数据库内部错误也是引发操作失败的原因。在上述所有情况下，操作将不起作用，且数据库 API 将抛出异常。

应用程序可在单个事务中执行一系列语句以及数据库操作，这样，如果任何语句或操作抛出异常，便不会应用这套操作中的任何数据库操作。应用程序使用 Python 函数定义要在事务中执行的操作，然后以该函数为参数调用 db.run_in_transaction()：

```
from google.appengine.ext import db

class Accumulator(db.Model):
  counter = db.IntegerProperty()

def increment_counter(key, amount):
  obj = db.get(key)
  obj.counter += amount
  obj.put()

q = db.GqlQuery("SELECT * FROM Accumulator")
acc = q.get()
```

db.run_in_transaction(increment_counter, acc.key(), 5)

db.run_in_transaction() 使用函数对象、位置以及要传递给该函数的关键字参数。如果该函数返回了一个值，那么，db.run_in_transaction() 也将返回该值。

如果该函数返回，则提交事务，并应用数据库操作的所有作用。如果该函数抛出异常，则事务被 [回滚]，并且不会应用任何作用。

如果函数抛出 Rollback 异常，db.run_in_transaction() 将返回 None。对于其他任何异常，db.run_in_transaction() 将重新抛出该异常。

事务的功能

数据库对单个事务中可完成的功能施加了许多限制。

事务中的所有数据库操作必须在同一实体组中的实体上进行。这包括 db.get()、put() 和 delete()。请注意，每个根实体都属于单独的实体组，因此，单个事务不能创建多个根实体或在多个根实体上进行操作。有关实体组的说明，请参阅键和实体组。

事务不能使用 Query 或 GqlQuery 执行查询。但是，事务可使用键和 db.get() 检索数据库实体。Key 可传递给事务函数，或者在该函数中构建，包括指定键名或 ID 以及 Key.from_path()、Model.get_by_key_name() 或 Model.get_by_id()。

应用程序不能在单个事务中多次创建或更新实体。

注意：截止到本文档编写之时，仍存在一个问题使用户无法在单个事务中创建新的根实体及其子孙实体。在此问题得到解决之前，根实体和子孙实体必须在不同的事务中创建。

事务函数中允许使用其他所有 Python 代码。除了数据库操作，该事务函数不应具有副作用。如果某数据库操作因其他用户同时在实体组中更新实体而失败，该事务函数则可进行多次调用。如果发生这种 情况，数据库 API 将以固定次数重新尝试调用该事务。如果以上操作全部失败，db.run_in_transaction() 将抛出 TransactionFailedError。

同样，事务函数不应具有由该事务成功与否所决定的副作用，除非调用此事务函数的代码明确如何撤消这些作用。例如，如果事务存储一个新数据库实体并保 存该创建的实体的 ID 以供以后使用，然后该事务失败，则保存的 ID 无法引用预期的实体，因为该实体创建被回滚。在这种情况下，调用代码一定要谨慎，切勿使用已保存的 ID。

事务的用途

以上示例说明了事务的一个用途：使用与属性当前值相关的新属性值更新实体。

```
def increment_counter(key, amount):
    obj= db.get(key)
    obj.counter+= amount
```

```
    obj.put()
```

这需要使用事务，因为在该用户的请求调用 db.get(key) 之后和调用 obj.put() 之前，该值可能会被另一用户更新。如果不使用事务，该用户的请求将使用更新前的 obj.counter 值，且 obj.put() 将覆盖此更新。如果使用事务，则可保证实体不会在两次调用之间被更改。如果实体在事务期间进行了更新，则该事务会重新调用，直到所有步骤都在不中断地情况下完成。

事务的另一常见用途是使用已命名的键更新实体，或当实体不存在时创建实体：

```
class SalesAccount(db.Model):
    address= db.PostalAddressProperty()
    phone_number= db.PhoneNumberProperty()

def create_or_update(parent_obj, account_id, address, phone_number):
    obj= db.get(Key.from_path("SalesAccount", account_id, parent=parent_obj))
  if not obj:
        obj= SalesAccount(parent=parent_obj,
                                    address=address,
                                    phone_number=phone_number)
  else:
        obj.address= address
        obj.phone_number= phone_number

    obj.put()
```

同以前一样，如果其他用户尝试使用相同的 account_id 创建或更新实体时，必须使用事务处理这种情况。如果不使用事务，则当实体不存在，而两个用户均尝试创建该实体时，第二个用户将失败。如果使用事务，将重试第二个用户的尝试（请注意，现在该实体已存在）并将更新该实体。

创建或更新非常有用，以致存在一个针对它的内置方法：Model.get_or_insert()，该方法的参数为键名、可选的父项以及传递给 Model 构造函数的参数（在该名称和路径的实体不存在的情况下）。获取尝试和创建会在同一事务中进行，因此（如果该事务成功完成）该方法会始终返回表示实际实体的 Model 实例。

提示：应当尽可能快地执行事务，以减少该事务所用实体被更改从而需要重试该事务的可能性。在该事务以外尽 可能多地准备数据，然后执行该事务的数据库操作，该操作要求数据处于稳定状态。应用程序应当为在事务中所使用的对象准备 Key，然后使用 db.get() 以抓取该事务中的实体。

Model 类

Model 类是数据模型定义的超类。

Model 由 google.appengine.ext.db 模块提供。

- 简介

简介

通过定义将 Model 作为子类的类，应用程序可以定义数据模型。使用 Model 属性和 Property 类实例，可以定义 Model 的属性。例如：

```
class Story(db.Model):
  title= db.StringProperty()
  body= db.TextProperty()
  created= db.DateTimeProperty(auto_now_add=True)
```

通过实例化 Model 类的子类，应用程序可以创建新数据实体。实体的属性可以使用实例的属性来分配，也可以作为构造函数的关键字参数。

```
s= Story()
s.title= "The Three Little Pigs"

s= Story(title="The Three Little Pigs")
```

Model 子类的名称会被用作数据库实体类型的名称。属性的名称会被用作实体上对应属性的名称。名称以下划线 (_) 开头的 Model 实例属性会被忽略，因此您的应用程序可以使用这些属性来将尚未保存到数据库中的数据存储到 Model 实例上。

数据库和 Model 类 API 可以对属性名称和 Model 实例属性实施若干限制。有关完整说明，请参阅禁止使用的属性名称。

数据实体可以有可选的父实体。父子关系可以形成实体组，用于控制数据库中的事务性和数据位置。通过将父实体传递到子实体的构造函数（作为 parent 参数），应用程序可以创建两个实体之间的父子关系。有关父和祖先的详细信息，请参阅键和实体组。

每个实体都有一个键，该键是代表实体的唯一标识符。实体可以有一个可选的键名，该名称是指定类型的实体之间的唯一字符串。实体的类型和名称可以和 Key.from_path() 及 Model.get_by_key_name() 方法配合使用，以检索实体。有关键的详细信息，请参阅键和实体组。

Model.get_or_insert() 方法可以用于检索可能不存在的实体，并在必要的情况下在数据库中创建该实体。

```
keyname = "some_key"

s = Story.get_or_insert(keyname, title="The Three Little Pigs")
```

注意：在第一次 put() Model 实例之前（明确地执行该操作或通过 Model.get_or_insert() 执行），Model 实例在数据库中并没有对应的实体。

Model 类由 google.appengine.ext.db 包提供。

构造函数

Model 类的构造函数定义如下：

```
class Model(parent=None, key_name=None, **kw)
```

数据模型定义的超类。

参数：

parent
: 作为新实体的父实体的 Model 实例。

key_name
: 新实体的名称。该名称会成为主键的一部分。如果为 None，将为键使用系统生成的 ID。

key_name 的值不得以数字开头 也绝不能采用 __*__ 的形式。如果您的应用程序使用用户提交的数据来作为数据库实体的键名（例如电子邮件地址），则应用程序应先清理该值（例如为其添加 [key:] 等已知的字符串前缀），使其符合这些要求。

key_name 存储为 Unicode 字符串，str 值转换为 ASCII 文本。

**kw
: 实例的属性的初始值，作为关键字参数。每个名称都与 Model 类中定义的属性对应。

类方法

Model 类提供以下类方法：

Model.get(keys)

获取指定 Key 对象的 Model 实例。键必须代表 Model 类型的实例。如果提供的键类型不正确，就会抛出 KindError。

该方法类似于 db.get() 函数，但具有额外的类型检查。

参数：

keys
: Key 对象或 Key 对象的列表。还可以是 Key 对象的字符串版本，或字符串列表。

Model.get_by_id(ids, parent=None)

获取指定数字 ID 的 Model 实例。

参数：

ids
: 数字实体 ID，或数字实体 ID 列表。

parent
: 所请求实体的父实体，是 Model 实例或 Key 实例，如果所请求实体没有父实体，则为 None（默认）。一个调用所请求的多个实体必须全部具有相同的父实体。

如果 ids 是代表一个名称的字符串，则该方法会返回该名称的 Model 实例，如果实体不存在，则返回 None。如果 ids 是一个列表，则该方法会返回 Model 实例的列表，如果对应的 Key 不存在实体，则返回 None 值。

Model.get_by_key_name(key_names, parent=None)

获取指定键名的 Model 实例。

参数：

key_names
: 键名或键名列表。

parent
: 所请求实体的父实体，是 Model 实例或 Key 实例，如果所请求实体没有父实体，则为 None（默认）。一个调用请求的多个实体必须全部具有相同的父实体。

如果 key_names 是代表一个名称的字符串，则该方法会返回该名称的 Model 实例，如果实体不存在，则返回 None。如果 key_names 是一个列表，则该方法会返回 Model 实例的列表，如果对应的 Key 不存在实体，则返回 None 值。

Model.get_or_insert(key_name, **kwds)

使用一个事务获取或创建具有指定键名的 Model 类型的实体。如果两个用户同时尝试使用指定的名称来获取或插入实体，则该事务可以确保两个用户都能拥有指向该实体的 Model 实例，而不管创建实体的过程如何。

参数：

key_name
: 实体的键名

**kwds
: 要传递给 Model 类的关键字参数（当具有指定键名的实例不存在时）。如果所需实体有父实体，则需要 parent 参数。

该方法会返回代表所请求实体的 Model 类的实例，与该实例是否存在或是否由方法创建无关。与所有数据库操作一样，如果无法完成事务，该方法会抛出 TransactionFailedError 错误。

Model.all()

返回代表与该 Model 对应的类型的所有实体的 Query 对象。在执行 Query 对象上的方法之前，可以对查询进行过滤和排序。有关详细信息，请参阅 Query。

Model.gql(query_string, *args, **kwds)

对该 Model 的实例执行 GQL 查询。

参数：

query_string
: GQL 查询 SELECT * FROM *model* 后的部分（使用该类方法时是暗含的）。

*args
: 位置参数绑定，类似于 GqlQuery 构造函数。

**kw
: 关键字参数绑定，类似于 GqlQuery 构造函数。

s= Story.gql("WHERE title = :1", "Little Red Riding Hood")

s= Story.gql("WHERE title = :title", title="Little Red Riding Hood")

返回值是 GqlQuery 对象，可以用于访问结果。

Model.kind()
: 返回 Model 的类型，通常是 Model 子类的名称。

Model.properties()
: 返回为该 Model 类定义的所有属性的参照表。

实例方法

Model 实例有以下方法：

key()

返回该 Model 实例的数据库 Key。

在 put() 入数据库之前，Model 实例没有键。在实例拥有键之前调用 key() 会抛出 NotSavedError 错误。

put()

将 Model 实例存储在数据库中。如果 Model 实例是新创建的并且之前从未存储过，则该方法会在数据库中创建新的数据实体。否则，该方法会用当前属性值更新数据实体。

该方法会返回存储的实体的 Key。

delete()

从数据库中删除 Model 实例。如果实例从未被 put() 到数据库，删除不会起任何作用。

is_saved()

如果 Model 实例至少已被 put() 到数据库中一次，则返回 True。

该方法只会检查自实例创建后是否至少已存储过一次。它不会检查自最后一次被 put() 后实例的属性是否更新过。

dynamic_properties()

返回针对该 Model 实例定义的所有动态属性的名称列表。此方法仅适用于 Expando 类的实例。对于非 Expando 模型实例，此方法会返回空列表。

parent()

返回该实例的父实体的 Model 实例，如果该实例没有父实体则返回 None。

parent_key()

返回该实例的父实体的 Key，如果该实例没有父实体则返回 None。

to_xml()

返回 Model 实例的 XML 表示方法。

属性值符合 Atom 和 GData 规范。

禁止使用的属性名称

数据库及其 API 对实体属性名称和 Model 实例属性实施若干限制。

数据库保留所有以两根下划线 (__*__) 开头和结尾的属性名称。数据库实体不能有此类名称的属性。

Python 模型 API 会忽略 Model 或 Expando 上所有以下划线 (_) 开头的属性。应用程序可以使用这些属性将数据与未保存到数据库的 Model 对象相关联。

最后，Python 模型 API 可以使用对象属性定义 Model 的属性，并且在默认情况下，数据库实体属性会根据这些属性进行命名。由于 Model 类具有用于其他目的的若干种属性和方法，因此这些属性无法用于 Python API 中的属性。例如，Model 无法使用属性 key 来访问属性。

但是，通过给属性构造函数赋予 name 参数，属性可以为数据库指定与属性名称不同的其它名称。这样，数据库实体可以拥有与 Model 类中保留的属性类似的属性名称，并可在该类中使用不同的属性名称。

```
class MyModel(db.Model):
  obj_key= db.StringProperty(name="key")
```

以下属性名称为 Model 类在 Python API 中所保留的属性名称：

- all
- app
- copy
- delete
- entity
- entity_type
- fields
- from_entity
- get
- gql
- instance_properties
- is_saved
- key
- key_name
- kind
- parent
- parent_key
- properties
- put
- setdefault
- to_xml
- update

Expando 类

Expando 类是用于数据模型定义（其属性动态确定）的超类。Expando 模型可以具有固定属性（类似于 Model）和动态属性（运行时分配给实体）的组合。

Expando 由 google.appengine.ext.db 模块提供。

Expando 是 Model 的子类，且从该类继承它的类和实例方法。Expando 类不定义或覆盖任何方法。

简介

Expando 模型可以有固定属性和动态属性。固定属性的行为类似 Model 的属性，并在 Expando Model 类中使用类属性以相似的方式定义。当在实例上为动态属性分配值时创建动态属性。同一 Expando 类的两个实例可拥有不同组的动态属性，甚至可以有名称相同但类型不同的动态属性。动态属性始终是可选的且没有默认值：直到为这些属性分配值，它们才存在。

动态属性无法使用 Property 实例来执行验证、设置默认值或对值应用自动逻辑。动态属性只是存储支持的数据库类型的值。请参阅类型和 Property 类。

动态属性与固定属性的不同之处还在于不能对类属性和数据库属性名称使用不同的名称。请参阅禁止使用的属性名称。

提示：如果您要使用 Property 类验证动态属性值，您可以实例化该 Property 类并对该值调用其 validate() 方法。

Expando 子类可以定义类似 Model 类的固定属性。Expando 的固定属性的行为与 Model 的属性类似。Expando 实例可以拥有固定属性和动态属性。

```
import datetime

class Song(db.Expando):
  title= db.StringProperty()

crazy= Song(title='Crazy like a diamond',
            author='Lucy Sky',
            publish_date='yesterday',
            rating=5.0)

hoboken= Song(title='The man from Hoboken',
              author=['Anthony', 'Lou'],
              publish_date=datetime.datetime(1977, 5, 3))

crazy.last_minute_note=db.Text('Get a train to the station.')
```

可以删除 Expando 实例的动态（非固定）属性。要删除动态属性，应用程序将删除实例的属性：

```
del myobj.myprop
```

构造函数

Model 类的构造函数定义如下：

class Expando parent=None, key_name=None, **kw)

一个 Model 类，其属性在使用前不需要在类中定义。与 Model 类似，Expando 类必须成为子类才能定义数据实体的类型。

Expando 是 Model 的子类，且可以继承或覆盖其方法。

参数：

parent
: 新实体的父实体的 Model 实例或 Key 实例。

key_name

新实体的名称。该名称会成为主键的一部分。如果为 None，将为键使用系统生成的 ID。

key_name 的值不得以数字开头 也绝不能采用 __*__ 的形式。如果您的应用程序使用用户提交的数据作为实体键名（例如电子邮件地址），应用程序应先通过用已知的字符串（例如 [key:]）作为前缀来清理值以满足这些要求。

key_name 存储为 Unicode 字符串，str 值转换为 ASCII 文本。

**kw
: 实例的属性的初始值，作为关键字参数。每个名称与新实例的属性对应，并可能与 Expando 类中定义的固定属性对应，或与动态属性对应。

Property 类

Property 类是数据模型的属性定义的超类。Property 类可以定义属性值的类型、值的验证方式以及值在数据库中的存储方式。

Property 由 google.appengine.ext.db 模块提供。

- 简介
- Property()
- 类属性：
  - Property.data_type
- 实例方法：
  - default_value()
  - validate()
  - empty()
  - get_value_for_datastore()
  - make_value_from_datastore()

简介

Property 类介绍了值类型、默认值、验证逻辑以及 Model 属性的其他功能。每个 Property 类都是 Property 类的子类。数据库 API 包含每个数据库值类型的 Property 类，以及提供除数据库类型以外的其他功能的多个 Property 类。请参阅类型和 Property 类。

Property 类可接受通过传递到构造函数的参数所进行的配置。基类构造函数支持通常在所有 Property 类中都受支持的多个参数，包括在数据库 API 中提供的所有参数。此类配置可以包含默认值（无论是否需要确切值）、可接受的值的列表以及自定义的验证逻辑。有关配置该属性的详细信息，请参阅特定属性类 型的文档。

Property 类定义数据库属性的模型。它不包含 Model 实例的属性值。Property 类的实例属于 Model 类，而不是该类的实例。在 Python 术语中，Property 类实例是可以自定义 Model 实例属性的行为方式的 [描述符]。有关描述符的详细信息，请参阅 Python 文档。

构造函数

Property 基类的构造函数定义如下：

class Property(verbose_name=None, name=None, default=None, required=False, validator=None, choices=None)

Model 属性定义的超类。

参数：

verbose_name
: 用户友好的属性名称。该参数必须始终是属性构造函数的第一个参数。 djangoforms 库使用该参数来为表格字段标记标签，其他库可使用该参数进行类似的操作。

name
: 属性的存储名称，可在查询中使用。默认情况下，该名称是用于属性的属性名称。由于 Model 类具有属性以外的属性（无法用于属性），因此属性可以使用 name 来使用保留的属性名称来作为数据库中的属性名称，并使用其他名称作为属性名称。有关详细信息，请参阅禁止使用的属性名称。

default

属性的默认值。如果属性值从未指定过值或指定的值为 None，则该属性值会被视为默认值。

注意：Model 类定义随应用程序代码的其余部分一起缓存。其中包括缓存属性的默认值。请勿通过请求特定的数据（例如 users.get_current_user()）在 Model 定义中设置 default。而应为初始化属性值的 Model 类定义 __init__() 方法。

required

如果为 True，则属性值不能为 None。Model 实例必须通过其构造函数初始化所有必需的属性，这样创建实例时才不会缺少值。如果在没有初始化必需属性的情况下尝试创建实例，或者尝试将 None 分配给必需的属性，则会抛出 BadValueError 错误。

如果在构造函数中没有为必需并具有默认值的属性赋值，则该属性会使用默认值。但是，不能为该属性分配 None 值，也不能在分配了其他值后自动恢复默认值。您随时可以访问属性的 default 属性来获取该值并对其进行显式分配。

validator
: 分配属性值时应调用便以验证该值的函数。该函数使用该属性值作为其唯一的参数，如果属性值无效，则会抛出异常错误。在执行了其他验证（例如检查必需的属性是否具有值）之后，便会调用指定的验证程序。

choices
: 可接受的属性值的列表。如果设置了该参数，则不能给属性分配该列表以外的其它值。与 required 和其他验证一样，Model 实例必须初始化选择的所有属性，这样才不会使用无效值创建实例。如果 choices 为 None，则以其他方式通过验证的所有值都可以接受。

类属性

Property 类的子类定义以下类属性：

data_type
: 属性接受作为 Python 自有值的 Python 数据类型或类。

实例方法

Property 类实例具有以下方法：

default_value()

返回属性的默认值。基础实施方案使用传递到构造函数 default 参数的值。Property 类可以替换该方法以提供特殊的默认值行为，例如 DateTimeProperty 的 auto-now 功能。

validate(value)

属性的完整验证程序。如果 value 有效，则会返回该值（保持不变或根据必需的类型进行了改变）。否则，便会抛出相应的异常错误。

基础实施方案会检查以下内容：如有需要，value 是否为 None（传递到基础 Property 构造函数的 required 参数）；如果已根据选择的内容对属性进行了配置，该值是否为一个有效的选择（choices 参数）；如果存在，该值是否通过自定义验证程序的验证（validator 参数）。

使用该属性类型的 Model 进行实例化（使用默认值或初始化值）以及为该类型的属性进行赋值时，会调用该验证程序。该程序不应该有副作用。

empty(value)

如果该属性类型的 value 使用空值，则返回 True。基础实施方案与 not value 等效，足以适用于大多数类型。其他

类型（如布尔值类型）可以使用更合适的测试来替换该方法。

get_value_for_datastore(model_instance)

返 回应该为指定 Model 实例中的该属性保存到数据库中的值。基础实施方案只会返回 Model 实例中该属性的 Python 自有值。Property 类可以将其替换，以便数据库能够使用与 Model 实例不同的其他数据类型，或在存储 Model 实例之前执行其他数据转换。

make_value_from_datastore(value)

从数据库中返回指定值的 Python 自有表达方式。基础实施方案只返回该值。Property 类可以将其替换，以便 Model 实例能够使用与数据库不同的其他数据类型。

Query 类

Query 类是一个数据库查询接口，可以使用对象和方法来准备查询。

Query 由 google.appengine.ext.db 模块提供。

简介

应用程序可通过调用具有 Model 类（其实体为 Query 对象）的构造函数或调用该类的 all() 类方法来创建 Query 对象。

```
class Song(db.Model):
  title= db.StringProperty()
  composer= db.StringProperty()
  date= db.DateTimeProperty()

query= db.Query(Song)

query= Song.all()
```

如果未作修改，该对象表示对指定类型的所有实体的查询。方法调用会使用属性条件 (filter())、祖先条件 (ancestor())

和排序 (order()) 自定义查询。为了方便起见，这些方法会返回 self，这样就可将它们串联成一个语句。

```
query.filter('title =', 'Imagine')
query.order('-date')
query.ancestor(key)

query.filter('title =', 'Imagine').order('-date').ancestor(key)
```

应用程序使用以下两种方法之一执行该查询：

通过调用 fetch() 的方法。此方法可执行对数据库的单次调用以抓取结果，结果不能超过指定数量。 Query 对象不对结果进行缓存，因此，再次调用 fetch() 会重新执行该查询。

```
results= query.fetch(limit=5)for songin results:
  print song.title
```

通 过将 Query 对象视为可迭代对象的方法。迭代程序可以从数据库中小批量地检索结果，从而允许应用程序停止迭代结果，避免抓取超出所需数量的结果。当检索出与该查询相匹 配的所有结果时，迭代会停止。正如调用 fetch() 一样，迭代程序接口不对结果进行缓存，因此，从 Query 对象中创建新迭代程序会重新执行该查询。

```
for songin query:
  print song.title
```

另请参阅 GqlQuery，这是使用一种类似 SQL 查询语言的 Query 类。

注意：基于索引的数据结构以及支持数据库查询的算法不支持某些类型的查询。有关详细信息，请参阅查询和索引：对查询的限制。

构造函数

Query 类的构造函数的定义如下：

```
class Query(model_class)
```

使用对象和方法准备查询的数据库查询接口。

由构造函数返回的 Query 实例表示对该类型的所有实体的查询。实例方法 filter()、order() 和 ancestor() 将标准应用到该查询以对结果进行过滤或排序。

参数：

model_class
代表查询的数据库实体类型的 Model（或 Expando）类

实例方法

Query 类具有以下几种实例方法：

filter(property_operator, value)

将属性条件过滤器添加至该查询。该查询只会返回具备满足所有条件的属性的实体。

参数：

property_operator
包含属性名称和比较运算符的字符串。支持以下比较运算符：< <= = >= >（不支持不等于 (!=) 和 IN 运算符）。

value
比较过程中所用的位于表达式右侧的值。其类型应当为被比较属性的值数据类型。请参阅类型和 Property 类。

```
query.filter('height >', 42).filter('city = ', 'Seattle')

query.filter('user = ', users.get_current_user())
```

order(property)

为结果添加排序。结果将根据首先添加的顺序进行排列。

参数：

property
一个字符串，要为其排序的属性的名称。要将排列顺序指定为降序，请在名称前加一个连字符 (-)。不加连字符，排列顺序将为升序。

```
# Order by last name, alphabetical:
query.order('last_name')
# Order tallest to shortest:
query.order('-height')
```

ancestor(ancestor)

将祖先条件过滤器添加至该查询。该查询只会返回以指定实体作为祖先（在其路径中的任何位置）的那些实体。

参数：

ancestor
代表该祖先的 Model 实例或 Key 实例。

get()

执行该查询，然后返回第一个结果，如果该查询未返回任何结果，则返回 None。

get() 暗含 [限制] 为 1。最多从数据库中抓取 1 个结果。

fetch(limit, offset=0)

执行该查询，然后返回结果。

limit 和 offset 参数控制从数据库抓取的结果数量，以及通过 fetch() 方法返回的结果数量：

数据库会为应用程序抓取 offset + limit 个结果到应用程序。数据库本身不会跳过第一个 offset 结果。
fetch() 方法则会跳过前 offset 个结果，然后返回剩余结果（limit 个结果）。
该查询具有与 offset 加 limit 数量之和成线性对应关系的性能特征。

注意：fetch() 返回最多 1000 个结果。如果有超过 1000 个实体与查询相匹配，且并未指定任何限制或使用了大于 1000 的限制，则 fetch() 仅返回前 1000 个结果。

参数：

limit

要返回的结果的数量。如果满足条件的结果数量不足，则返回的结果可能要少于 limit 个。

limit 是必需的参数。要在结果数量未知时获取查询的每个结果，可将 Query 对象用作可迭代对象，而不要使用 fetch() 方法。

offset

要跳过的结果的数量。

返回值是一个 Model 实例列表，可能是一个空列表。

count(limit)

返回该查询抓取的结果的数量。

count() 比通过常量系数检索所有数据要快一些，但是运行时间仍随结果集大小而增加。如果预期的数量很少，或指定了一个 limit，那么，最好只使用 count()。

注意：count() 返回的最大值为 1000。如果与查询条件相匹配的实体的实际数量超出了最大值，count() 会只返回 1000 个结果。

参数：

limit

要计数的结果的最大数量。

GqlQuery 类

GqlQuery 类是一种使用 App Engine 查询语言 GQL 的数据库查询接口。

GqlQuery 由 google.appengine.ext.db 模块提供。

简介

GQL 是一种类似于 SQL 的查询语言，适用于查询 App Engine 数据库。有关 GQL 语法和功能的完整讨论，请参阅 GQL 参考。

GqlQuery 构造构造函数采用以 SELECT * FROM *model-name* 开头的完整 GQL 语句作为参数。WHERE 子句中的值可以是字符串或数字字母，或可以使用值的参数绑定。绑定参数最初可以使用位置或关键字参数绑定到构造函数。

```
query= GqlQuery("SELECT * FROM Song WHERE composer = 'Lennon, John'")

query= GqlQuery("SELECT * FROM Song WHERE composer = :1", "Lennon, John")

query= GqlQuery("SELECT * FROM Song WHERE composer = :composer", composer="Lennon, John")
```

为了方便起见，Model 和 Expando 类有种可返回 GqlQuery 实例的 gql() 方法。这种方法在不使用 SELECT * FROM *model-name* 的情况下采用 GQL 查询字符串，这是暗含的。

```
query= Song.gql("WHERE composer = 'Lennon, John'")
```

与使用 Query 类一样，应用程序通过调用 fetch() 方法或通过将 GqlQuery 对象视为可迭代来执行查询和访问结果。有关详细信息，请参阅 Query 文档。

Query 和 GqlQuery 访问结果的方式之间有一个不同之处：如果 GQL 查询包括一个 LIMIT 子句或一个 OFFSET 子句，将采用等效 fetch() 方法检索结果，即使迭代器接口用于访问结果也是如此。当某个 GqlQuery（其 GQL 包含 LIMIT 或 OFFSET）被作为可迭代使用时，将对数据库进行一次调用以抓取所有结果，然后迭代器从内存返回每个结果。

```
for songin q:
  print song.title
```

另请参阅 Query，一种使用对象和方法而不是 GQL 来准备查询的 Query 类。

注意： 支持数据库查询的基于索引的数据结构和算法不支持某些种类的查询。有关详细信息，请参阅 查询和索引：对查询的限制。

构造函数

GqlQuery 类的构造函数如下定义：

class GqlQuery(query_string, *args, **kwds)

使用 App Engine 查询语言 GQL 的 Query 对象。

参数：

query_string
: 以 SELECT * FROM *model-name* 开头的完整 GQL 语句。

*args
: 位置参数绑定。

**kwds
: 关键字参数绑定。

实例方法

GqlQuery 实例有以下方法：

bind(*args, **kwds)

重新绑定参数以进行查询。新查询将在重新绑定参数后第一次访问结果时执行。

重复使用带有新参数的 GqlQuery 对象比构建新的 GqlQuery 对象更快，因为重新绑定不需要再次解析查询字符串。

参数：

*args
: 新位置参数绑定。

**kwds
: 新关键字参数绑定。

get()

执行查询，然后返回第一个结果，或如果查询没有返回结果则返回 None。

get() 暗含 [limit] 为 1，并覆盖 GQL 查询的 LIMIT 子句（如果有）。最多从数据库中抓取 1 个结果。

fetch(limit, offset=0)

执行查询，然后返回结果。

limit 和 offset 参数控制从数据库抓取的结果数量，以及通过 fetch() 方法返回的结果数量：

- 数据库会抓取 offset + limit 个结果到应用程序。数据库本身不会跳过前 offset 个结果。
- fetch() 方法则会跳过前 offset 个结果，然后返回剩余结果（limit 个结果）。
- 该查询具有与 offset 加 limit 数量之和成线性对应关系的性能特征。

注意：fetch() 返回最多 1000 个结果。如果有超过 1000 个实体与查询相匹配，且并未指定任何限制或使用了大于 1000 的限制，则 fetch() 仅返回前 1000 个结果。

参数：

limit

要返回的结果的数量。该值覆盖 CQL 查询语句中的 LIMIT 子句（如果有）。如果没有足够的符合条件的可用结果，则可能被返回少于 limit 的结果。

limit 是必需参数。当结果数未知时，可迭代地使用 GqlQuery 对象而不是使用 fetch() 方法从查询获取每个结果。

offset

要跳过的结果的数量。该值覆盖 CQL 查询语句中的 OFFSET 子句或 LIMIT 子句中的偏移（如果有）。

返回值是一个 Model 实例列表，可能是一个空列表。

count(limit)

返回该查询抓取的结果的数量。

count() 比通过常量系数检索所有数据要快一些，但是运行时间仍随结果集大小而增加。如果预期的数量很少，或指定了一个 limit，那么，最好只使用 count()。

注意：count() 返回的最大值为 1000。如果与查询条件相匹配的实体的实际数量超出了最大值，count() 会只返回 1000 个结果。

参数：

limit

要计数的结果的最大数量。该值覆盖 CQL 查询语句中的 LIMIT 子句（如果有）。

Key 类

Key 类的实例代表数据库实体的唯一键。

Key 由 google.appengine.ext.db 模块提供。

简介

存储在数据库中的每个 Model 实例都拥有一个代表该对象的唯一键。Model 实例的 key() 方法会返回实例的 Key 对象。如果该实例从未被 put() 到数据库中，则 key() 方法会抛出 NotSavedError 错误。

应用程序可以使用 get() 函数检索指定键的 Model 实例。

Key 实例可以是数据库实体属性的值，包括 Expando 动态属性和 ListProperty 成员。ReferenceProperty 模型可以为 Key 属性值提供自动取消引用等各项功能。

构造函数

class Key(encoded=None)

数据库对象的唯一键。

通过将 Key 对象传递到 str()（或调用对象的 __str__() 方法），可以把键编码成字符串。编码为字符串的键是一个使用能安全地加在网址中的字符的不透明值。通过将编码为字符串的键传递到 Key 构造函数（encoded 参数），可以将其转换回 Key 对象

注意：编码为字符串的键可以转换回原始键数据。这样，当知道一个键时猜测其他键会更为容易。虽然编码为字符串的键值能够安全地加到网址中，但是只有在键的可猜测度不成为问题的情况下，应用程序才能这样做。

encoded

可以转换回 Key 的 Key 实例的 str 形式。

类方法

Key 类提供以下类方法：

Key.from_path(*args, **kwds)

从一个或多个实体键的祖先路径构建新的 Key 对象。

路径代表实体的父子关系的层次结构。路径中的每个实体都由实体的类型，以及其数字 ID 或键名来代表。完整路径代表路径中最后显示的实体，其祖先（父）作为前续实体。

例如，以下调用可以为类型为 Address 的实体创建一个数字 ID 为 9876 的键，其父实体的类型为 User，键名为 'Boris'：

k = Key.from_path('User', 'Boris', 'Address', 9876)

参数：

*args
从根实体到主题的路径。路径中的每个实体都由列表中的两个元素来表示：类型名称和该类型实体的键名或 ID。

**kwds
关键字参数。该方法支持一个关键字参数 (parent)，该参数可以指定添加到指定路径前面的父实体。其值是父实体的 Key。

有关路径的详细信息，请参阅键和实体组。

实例方法

Key 实例有以下方法：

app()

返回存储数据实体的应用程序的名称。

kind()

以字符串形式返回数据实体的类型。

id()

以整数形式返回数据实体的数字 ID，如果实体没有数字 ID 则返回 None。

name()

返回数据实体的名称，如果实体没有名称则返回 None。

id_or_name()

返回数据实体的名称或数字 ID（无论有哪个），如果实体既没有名称也没有数字 ID 则返回 None。

has_id_or_name()

如果实体有名称或数字 ID 则返回 True。

parent()

返回数据实体的父实体的 Key，如果该实体没有父实体则返回 None。

函数

google.appengine.ext.db 包提供以下函数：

get(keys)

获取任何 Model 的指定键的实体。

参数：

keys
: Key 对象或 Key 对象的列表。

如果提供了一个 Key，则返回值是相应 Model 类的实例，或者是 None（如果指定键没有实体）。如果提供了 Key 列表，则返回值是 Model 实例的对应列表，并且在对应 Key 不存在实体时该列表会具有 None 值。

另请参阅 Model.get()。

put(models)

将一个或多个 Model 实例放置到数据库中。

参数：

models
: 要存储的 Model 实例或 Model 实例的列表。

如果指定了多个 Model 实例，它们可能存储于多个实体组中。在每个实体组中，所有属于该组的实体都将在一个事务中写入。请参阅键和实体组。

如果在操作过程中发生错误，则肯定会抛出异常，即便某些实体其实已经写入。如果调用中的实体跨越多个实体组就可能发生这种情况。如果调用返回而没有抛出异常，则表示所有实体都已成功写入。

返回与存储的 Model 实例对应的 Key 对象（如果指定了 Model 实例）或 Key 对象的列表（如果指定了实例列表）。

delete(models)

从数据库中删除一个或多个 Model 实例。

参数：

models
: 要删除的 Model 实例、实体的 Key，或 Model 实例列表或实体的 Key 列表。

如同 Model.put() 一样，如果指定了多个键，它们可能位于多个实体组中。在每个实体组中，所有属于该组的实体都将在一个事务中删除。请参阅键和实体组。

如果在操作过程中发生错误，则肯定会抛出异常，即便某些实体其实已经删除。如果调用中的键跨越多个实体组就可能发生这种情况。如果调用返回而没有抛出异常，则表示所有实体都已成功删除。

run_in_transaction(function, *args, **kwargs)

在一个事务中运行包含数据库更新的函数。如果代码在事务过程中抛出异常，则事务中进行的所有数据库更新都将回滚。

参数：

function
: 要在数据库事务中运行的函数。

*args
: 传递到函数的位置参数。

**kwargs
: 传递到函数的关键字参数。

如果函数返回一个值，run_in_transaction() 将该值返回到调用程序。

如果函数抛出异常，事务将回滚。如果函数抛出 Rollback 异常，该异常不会再次抛出。对于其他任何异常，会针对调用程序再次抛出异常。

数据库使用乐观锁定并重新尝试事务。如果无法提交函数准备的事务，run_in_transaction() 将再次调用函数，然后重复固定次数的重新尝试。由于可能针对一个事务调用多次事务函数，所以函数不应有副作用，包括对参数的修改。

例如，如果由于高冲突率而无法提交事务，将抛出 TransactionFailedError。

有关事务的详细信息，请参阅事务。

```
def decrement(key, amount=1):
    counter= db.get(key)
    counter.count-= amount
  if counter.count< 0:        # don't let the counter go negative
      raise db.Rollback()
    db.put(counter)

q= db.GqlQuery("SELECT * FROM Counter WHERE name = :1", "foo")
counter= q.get()
db.run_in_transaction(decrement, counter.key(), amount=5)
```

类型和 Property 类

App Engine 数据库支持数据实体上的一系列固定的属性值类型。Property 类可定义与基本值类型相互转换的新类型，并且，这些值类型可与 Expando 动态属性和 ListProperty 聚合属性模型直接配合使用。

下表介绍了一些 Property 类，其值直接对应于基本数据类型。这些值类型中的任何一个都可在 Expando 动态属性或 ListProperty 聚合类型中使用。

Property 类	值类型	排序顺序
StringProperty	str unicode	Unicode（将 str 作为 ASCII 处理）
BooleanProperty	bool	False < True
IntegerProperty	int long	数字
FloatProperty	float	数字
DateTimeProperty DateProperty TimeProperty	datetime.datetime	按时间顺序
ListProperty StringListProperty	受支持类型的 list	如果按升序，则从最小的元素排列；如果按降序，则从最大的元素排列
ReferenceProperty SelfReferenceProperty	db.Key	按照路径元素（类型、ID 或名称、类型、ID 或名称...）
UserProperty	users.User	按照电子邮件地址 (Unicode)
BlobProperty	db.Blob	（不可排序）
TextProperty	db.Text	（不可排序）
CategoryProperty	db.Category	Unicode
LinkProperty	db.Link	Unicode

EmailProperty	db.Email	Unicode
GeoPtProperty	db.GeoPt	先按纬度排序，再按经度排序
IMProperty	db.IM	Unicode
PhoneNumberProperty	db.PhoneNumber	Unicode
PostalAddressProperty	db.PostalAddress	Unicode
RatingProperty	db.Rating	数字

数据库值类型

数据库实体属性值可以是以下类型之一：有关可与 Model 定义配合使用的相应 Property 类的列表，请参阅上文。

除了 Python 标准类型和 users.User，此部分介绍的所有类都由 google.appengine.ext.db 模块提供。

str 或 unicode

短字符串值，长度少于 500 个字节。

str 值假定为用 ascii 编解码器进行编码的文本，且在存储前转换为 unicode 值。数据库会将该值作为 unicode 值返回。对于使用其他编解码器的短字符串，请使用 unicode 值。

短字符串由数据库编入索引，并可在过滤器和排序顺序中使用。对于长度超过 500 个字节的文本字符串（未编入索引），请使用 Text 实例。对于长度超过 500 个字节的非编码字节字符串（也未编入索引），请使用 Blob 实例。

Model 属性：StringProperty

bool

布尔值，为 True 或 False。

Model 属性：BooleanProperty

int 或 long

整数值。

在存储之前，Python int 值会被转换为 Python long 值。存储为 int 的值将返回为 long。

Model 属性：IntegerProperty

float

浮点值。

Model 属性：FloatProperty

datetime.datetime

日期和时间。请参阅日期时间模块文档。

如果 datetime 值具有 tzinfo 属性，则将转换为 UTC 时区进行存储。值作为 UTC 从数据库返回，且具有 None 的 tzinfo。如果某应用程序需要日期和时间值位于特定时区，则在更新该值时必须正确设置 tzinfo，并在访问该值时将值转换到该时区。

有些库使用 TZ 环境变量以控制应用到日期时间值的时区。App Engine 将此环境变量设置为 "UTC"。请注意，在应用程序中更改此变量将不会更改一些日期时间函数的行为，这是因为在 Python 代码之外无法看到对环境变量的更改。

如果您只是将值转换到一个特定时区以及从一个特定时区转换值，您可以使用自定义 datetime.tzinfo 从数据库转换值：

```
class Pacific_tzinfo(datetime_module.tzinfo):
 """Implementation of the Pacific timezone."""
 def utcoffset(self, dt):
    return datetime_module.timedelta(hours=-8) + self.dst(dt)

 def _FirstSunday(self, dt):
    """First Sunday on or after dt."""
    return dt+ datetime_module.timedelta(days=(6-dt.weekday()))

 def dst(self, dt):
    # 2 am on the second Sunday in March
    dst_start= self._FirstSunday(datetime_module.datetime(dt.year, 3, 8, 2))
    # 1 am on the first Sunday in November
    dst_end= self._FirstSunday(datetime_module.datetime(dt.year, 11, 1, 1))

    if dst_start<= dt.replace(tzinfo=None) < dst_end:
       return datetime_module.timedelta(hours=1)
    else:
       return datetime_module.timedelta(hours=0)

 def tzname(self, dt):
    if self.dst(dt) == datetime_module.timedelta(hours=0):
       return "PST"
    else:
       return "PDT"

pacific_time= utc_time.astimezone(Pacific_tzinfo())
```

请参阅日期时间模块文档（包括 datetime.tzinfo）。您也可参阅第三方模块 pytz，但您要注意，该 pytz 分发有许多文件。

DateTimeProperty Model Property 类包括一些功能（例如，自动使用存储 Model 实例的日期和时间的功能）。这些是该 Model 的功能，但在原始数据库值（例如在一个 Expando 动态属性中）上不可用。

Model 属性：DateTimeProperty、DateProperty、TimeProperty

list

值列表，其中每个值都是受支持的数据类型之一。请参阅实体和模型：列表。

当 list 被用作 Expando 动态属性的值时，不能是空列表。这是由列表值的存储方式引起的：当列表属性无项目时，该列表属性在数据库中无任何表示。您可以使用一个静态属性和 ListProperty 类以表示属性的空列表值。

Model 属性：ListProperty

db.Key

其他数据库实体的键。

```
m= Employee(name="Susan", key_name="susan5")
m.put()
e= Employee(name="Bob", manager=m.key())
e.put()

m_key= Key.from_path("Employee", "susan5")
e= Employee(name="Jennifer", manager=m_key)
```

Model 属性：ReferenceProperty、SelfReferenceProperty

users.User

具有 Google 帐户的用户。

Model 属性：UserProperty

class Blob(arg=None)

二进制数据，形式为字节字符串。这是内置 str 类型的子类。

Blob 属性未被编入索引，且无法在过滤器或排序顺序中使用。

Blob 用于二进制数据（例如图像）。它使用 str 值，但该值作为字节字符串存储，而且不会作为文本进行编码。为大型文本数据使用 Text 实例。

Model 属性：BlobProperty

```
class MyModel(db.Model):
  blob= db.BlobProperty()

m= MyModel()
m.blob= db.Blob(open("image.png").read())
```

class Text(arg=None, encoding=None)

长字符串。这是内置 unicode 类型的子类。

arg，unicode 或 str 值。如果 arg 为 str，那么将使用 encoding 所指定的编码对其进行解析，如果未指定任何编码，则使用 ascii。有关 encoding 可能的值的信息，请参阅标准编码列表。

与实体属性（其值为一简单的 str 或 unicode）不同，Text 属性的长度可超过 500 个字节。然而，Text 属性未被编入索引，且无法在过滤器或排序顺序中使用。

Model 属性：TextProperty

```
class MyModel(db.Model):
  text= db.TextProperty()

m= MyModel()
m.text= db.Text(u"kittens")

m.text= db.Text("kittens", encoding="latin-1")
```

class Category(tag)

类别或 [标签]。这是内置 unicode 类型的子类。

Model 属性：CategoryProperty

```
class MyModel(db.Model):
  category= db.CategoryProperty()

m= MyModel()
m.category= db.Category("kittens")
```

在 XML 中，这是一个 Atom category 元素。请参阅 Atom 规范。

class Email(email)

电子邮件地址。这是内置 unicode 类型的子类。

Property 类和 Value 类都不执行对电子邮件地址的验证，它们只存储值。

Model 属性：EmailProperty

```
class MyModel(db.Model):
  email_address= db.EmailProperty()

m= MyModel()
m.email_address= db.Email("larry@example.com")
```

在 XML 中，这是一个 gd:email 元素。请参阅 GData API 参考。

class GeoPt(lat, lon=None)

由浮点纬度和经度坐标表示的地理点。

Model 属性：GeoPtProperty

在 XML 中，这是一个 georss:point 元素。请参阅 georss.org。

class IM(protocol, address=None)

即时消息手写板。

protocol 是该即时消息服务的规范网址。一些可能的值：

协议	说明
sip	SIP/SIMPLE
xmpp	XMPP/Jabber
http://aim.com/	AIM
http://icq.com/	ICQ
http://talk.google.com/	Google Talk
http://messenger.msn.com/	MSN Messenger
http://messenger.yahoo.com/	Yahoo Messenger
http://sametime.com/	Lotus Sametime
http://gadu-gadu.pl/	Gadu-Gadu
未知	未知或未指定

address 是手写板的地址。

Model 属性：IMProperty

```
class MyModel(db.Model):
  im= db.IMProperty()

m= MyModel()
```

```
m.im= db.IM("http://example.com/", "Larry97")
```

在 XML 中，这是一个 gd:im 元素。请参阅 GData API 参考。

class Link(link)

完全限定网址。这是内置 unicode 类型的子类。

Model 属性：LinkProperty

```
class MyModel(db.Model):
  link= db.LinkProperty()

m= MyModel()
m.link= db.Link("http://www.google.com/")
```

在 XML 中，这是一个 Atom link 元素。请参阅 Atom 规范。

class PhoneNumber(phone)

可读的电话号码。这是内置 unicode 类型的子类。

Model 属性：PhoneNumberProperty

```
class MyModel(db.Model):
  phone= db.PhoneNumberProperty()

m= MyModel()
m.phone= db.PhoneNumber("1 (206) 555-1212")
```

在 XML 中，这是一个 gd.phoneNumber 元素。请参阅 GData API 参考。

class PostalAddress(address)

邮政地址。这是内置 unicode 类型的子类。

Model 属性：PostalAddressProperty

```
class MyModel(db.Model):
  address= db.PostalAddressProperty()

m= MyModel()
m.address= db.PostalAddress("1600 Ampitheater Pkwy., Mountain View, CA")
```

在 XML 中，这是一个 gd:postalAddress 元素。请参阅 GData API 参考。

class Rating(rating)

由用户提供的对某段内容的评分，评分将为 0 到 100 之间的整数。这是内置 long 类型的子类。该类将验证值是否为 0 到 100 之间的整数，如果值无效，将抛出 BadValueError。

Model 属性：RatingProperty

```
class MyModel(db.Model):
  rating= db.RatingProperty()

m= MyModel()
m.rating= db.Rating(97)
```

在 XML 中，这是一个 gd:rating 元素。请参阅 GData API 参考。

Property 类

由 google.appengine.ext.db 提供的所有 Model Property 类都是 Property 基类的子类，并支持基本构造函数的所有参数。有关这些参数的信息，请参阅基类文档。

google.appengine.ext.db 包将提供以下 Model Property 类：

class BlobProperty(...)

二进制数据属性。

Blob 数据为字节字符串。对于可能包含编码的文本数据，请使用 TextProperty。

值类型：Blob

class BooleanProperty(...)

布尔属性。

值类型：bool

class CategoryProperty(...)

类别或 [标签]，描述性的词或短语。

值类型：Category

class DateProperty(verbose_name=None, auto_now=False, auto_now_add=False, ...)

日期属性，无当日时间。有关详细信息，请参阅 DateTimeProperty。

值类型：datetime.date。该值将在内部转换为 datetime.datetime。

class DateTimeProperty(verbose_name=None, auto_now=False, auto_now_add=False, ...)

日期和时间属性。

如果 auto_now 为 True，则无论 Model 实例在何时存储于数据存储中，该属性值都会设置为当前时间，并覆盖该属性的上一值。这将有助于跟踪一个 Model 实例 [上次修改] 的日期和时间。

如果 auto_now_add 为 True，除非已为该属性分配值，否则将在第一次在数据库中存储 Model 实例时将属性值设置为当前时间。这将有助于存储一个 Model 实例 [创建] 的日期和时间。

日期时间值作为 UTC 时区存储，并使用 UTC 时区返回。有关如何管理时区的论述，请参阅 datetime.datetime。

值类型：datetime.datetime

class EmailProperty(...)

电子邮件地址。

Property 类和 Value 类都不执行对电子邮件地址的验证，它们只存储值。

值类型：Email

class FloatProperty(...)

浮点数字属性。

值类型：float

class GeoPtProperty(...)

由浮点纬度和经度坐标表示的地理点。

值类型：GeoPt

class IMProperty(...)

即时消息手写板。

值类型：IM

class IntegerProperty(...)

整数属性。

在存储之前，Python int 值会被转换为 Python long 值。作为 int 存储的值将作为 long 返回。

值类型：int 或 long

class LinkProperty(...)

完全限定网址。

值类型：Link

class ListProperty(item_type, verbose_name=None, default=None, ...)

指定为 item_type 的类型的值列表。

查询中，将列表属性与值进行比较会针对该列表成员执行测试：如果该值出现在列表中的任何位置，则执行 list_property = value 测试；如果该列表中的任何成员都小于指定的值，则执行 list_property < value 测试，以此类推。

一个查询无法比较两个列表值。如果不分别测试成员的每个元素，则将无法测试两个列表是否相同。

item_type 是列表中项目的类型，为 Python 类型或类。列表值中的所有项目必须为指定的类型。item_type 必须为数据库值类型中的一种，且不能为 list。请参阅上面的数据库值类型。

ListProperty 静态属性的值不能为 None。然而，可以是空列表。

提示：由于 ListProperty 聚合类型不使用 Property 类，因此，Property 类的一些功能（例如自动值和验证）将不会自动应用到该列表值的成员。如果您要使用 Property 类验证一个成员值，您可以实例化该类并在该值上调用该类的 validate() 方法。

default 是列表属性的默认值。如果为 None，该默认值将为空列表。列表属性可定义一个自定义 validator 以防止产生空列表。

有关 ListProperty 和列表值的详细信息，请参阅实体和模型。

值类型：具有零个或多个值的 Python list，其中每个值都属于配置的类型

class PhoneNumberProperty(...)

可读的电话号码。

*值类型：*PhoneNumber

class PostalAddressProperty(...)

邮政地址。

值类型：PostalAddress

class RatingProperty()

由用户提供的对某段内容的评分，评分将为 0 到 100 之间的整数。

值类型： Rating

class ReferenceProperty(reference_class=None, verbose_name=None, collection_name=None, ...)

对其他 Model 实例的引用。例如，引用可能指明在具有属性的 Model 和由该属性引用的 Model 之间所存在的多对一的关系。

reference_class 是正被引用的 Model 实例的 Model 类。如果已指定，那么，将只能为此属性分配该类的 Model 实例。如果为 None，则任何 Model 实例都能成为该属性的值。

collection_name 为该属性的名称，提供给值为引用该实体的所有实体的 Query 的引用 Model 类。如果未设置 collection_name，那么将使用 *modelname*_set（包含使用小写字母并添加了 [_set] 的 Model 名称）。

注意：如果同一模型中存在多个引用了相同 Model 类的属性，则必须设置 collection_name。否则，当生成默认名称时将抛出 DuplicatePropertyError。

ReferenceProperty 将会自动引用和解除引用作为属性值的 Model 实例：Model 实例可直接分配到 ReferenceProperty，且将使用该 Model 实例的键。可将 ReferenceProperty 值用作 Model 实例，且将抓取数据库实体以及当该数据库实体首次以这种方式使用时创建 Model 实例。未触及的引用属性不对不需要的数据进行查询。

```
class Author(db.Model):
  name= db.StringProperty()

class Story(db.Model):
  author= db.ReferenceProperty(Author)

story= db.get(story_key)
author_name= story.author.name
```

使用 Key 值时，引用属性值可能引用不存在的数据实体。如果从数据库中删除一个引用的实体，将不会更新对该实体的引用。应用程序可明确地对 ReferenceProperty 的值（在这里为 Key）进行 db.get() 操作以测试引用的实体是否存在。

删除实体不会删除 ReferenceProperty 所引用的实体。

请参阅该引用属性简介。

值类型：db.Key（请参阅上文）

class SelfReferenceProperty(verbose_name=None, collection_name=None, ...)

对属于同一类的其他 Model 实例的引用。请参阅 ReferenceProperty。

值类型：db.Key（请参阅上文）

class StringListProperty(verbose_name=None, default=None, ...)

与 Python str 或 unicode (basestring) 值的 ListProperty 相似。请参阅 ListProperty。

值类型：str 或 unicode 值的 Python list

class StringProperty(verbose_name=None, multiline=False, ...)

短字符串属性。使用 500 个字节或更少字节的 Python str 或 unicode (basestring) 值。

StringProperty 属性值已被编入索引，并可在过滤器和排序顺序中使用。

如果 multiline 为 False，那么该值不能包含换行字符。djangoforms 库可使用此值类型强制区分数据库模型中的文本字段和文本区域字段，而且，其他库可使用此值类型实现类似的目的。

值类型：str 或 unicode

class TextProperty()

长字符串。

与 StringProperty 不同，TextProperty 值的长度可超过 500 个字节。然而，TextProperty 值未被编入索引，且无法在过滤器或排序顺序中使用。

TextProperty 值存储包含文本编码的文本。对于二进制数据，请使用 BlobProperty。

值类型：Text

class TimeProperty(verbose_name=None, auto_now=False, auto_now_add=False, ...)

时间属性，无日期。使用 Python 标准库的 datetime.time 值。有关详细信息，请参阅 DateTimeProperty。

值类型：datetime.time 。该值将在内部转换为 datetime.datetime。

class UserProperty()

具有 Google 帐户的用户。

UserProperty 构造函数不接受默认值。

值类型：users.User（请参阅上文）

GQL 参考

GQL 是一种类似于 SQL 的语言，用于从 App Engine 可扩展数据库检索数据实体。虽然 GQL 的功能与用于传统

关系数据库的查询语言的功能不同，但 GQL 语法类似 SQL 的语法。

GQL 语法可以总结如下：

```
SELECT * FROM <kind>
    [WHERE <condition> [AND <condition> ...]]
    [ORDER BY <property> [ASC | DESC] [, <property> [ASC | DESC] ...]]
    [LIMIT [<offset>,]<count>]
    [OFFSET <offset>]

<condition> := <property> {< | <= | > | >= | = | != } <value>
<condition> := <property> IN <list>
<condition> := ANCESTOR IS <entity or key>
```

与使用 SQL 一样，GQL 关键字不区分大小写。类型和属性名称区分大小写。

GQL 查询返回零个或更多请求的类型的数据实体，如同类型的 Model 类实例。结果始终是完整的实体，所以每个 GQL 查询始终以 SELECT * FROM 开头，后面带有类型的名称。（类似于 SQL 的字段指定符始终是 *。保留了 SQL 语法便于用户熟悉。）GQL 查询无法执行类似 SQL 的 [join] 查询。

可选 WHERE 子句对结果集进行过滤以得出符合一个或多个条件的实体。每个条件都使用比较运算符将实体的属性与值进行比较。如果用 AND 关键字指定了多个条件，则实体必须符合所有条件才能由查询返回。GQL 没有 OR 运算符。但是，它具有 IN 运算符，该运算符可提供一种受限形式的 OR。

IN 运算符将属性的值与列表中的每一项进行比较。IN 运算符等同于许多 = 查询（每个值一个）一起进行 OR 运算。查询可以返回指定属性的值等于列表中的任何值的实体。

注意：IN 和 != 运算符在后台使用多个查询。例如，IN 运算符对于列表中的每一项都执行单独的基层数据库查询。返回的实体是所有基层数据库查询的交叉产物的结果，并且消除了重复项。对于任一 GQL 查询，最多允许 30 个数据库查询。

条件还可以使用 ANCESTOR IS 操作符测试实体是否采用指定的实体作为祖先。值为祖先实体的 Model 实例或 Key。有关祖先的详细信息，请参阅键和实体组。

属性名称始终位于比较的左侧。右侧可以是以下内容之一（对应于属性的数据类型）：

- str 常量，形式为单引号括起的字符串。字符串中的单引号字符必须转义为 ''。例如：'Joe''s Diner'
- 整数或浮点数常量。例如：42.7
- 布尔常量，如 TRUE 或 FALSE。
- 绑定参数值。在查询字符串中，位置参数按数字引用：title = :1 关键字参数按名称引用：title = :mytitle

绑定参数可以绑定为传递到 GqlQuery 构造函数或 Model 类的 gql() 方法的位置参数或关键字参数。必须使用参数绑定来指定没有对应值常量语法的属性数据类型，包括列表数据类型。可以在 GqlQuery 实例的生命周期中使用 bind() 方法用新值重新绑定参数绑定（例如为了有效地重复使用查询）。

可选的 ORDER BY 子句指示应按照指定属性进行排序，以升序 (ASC) 或降序 (DESC) 返回结果。如果未指定方向，其默认为 ASC。ORDER BY 子句可以用逗号分隔列表的形式指定多个排序顺序，从左到右进行评估。

可选的 LIMIT 子句使查询在前 count 个实体后停止返回结果。LIMIT 还可以包括 offset 以跳过许多结果以便找到要返回的第一个结果。如果不使用 LIMIT 子句，可选的 OFFSET 子句可以指定 offset。

注意：LIMIT 子句最大为 1000。如果指定了大于最大值的限制，则将使用最大值。该最大值也适用于 GqlQuery 类的 fetch() 方法。

注意：与 fetch() 方法的 offset 参数类似，GQL 查询字符串中的 OFFSET 不会减少从数据库抓取的实体数。它仅影响 fetch() 方法返回哪些结果。具有偏移的查询具有与偏移大小成线性关系的执行特性。

有关执行 GQL 查询、绑定参数和访问结果的信息，请参阅 GqlQuery 类和 Model.gql() 类方法。

异常

google.appengine.ext.db 包提供以下 exception 类：

exception Error()

这是该包中所有异常的基类。

exception BadArgumentError()
    向查询方法提供了错误的参数。
exception BadFilterError()
    查询中的过滤条件字符串无效。
exception BadKeyError()
    提供的键字符串是无效的键。
exception BadPropertyError()
    无法创建属性，因为其名称不是字符串。
exception BadQueryError()
    查询字符串是无效的查询。
exception BadRequestError()
    对数据库服务的请求有一个或多个无效属性。这可能是因为 Model 的子类用错误的实现方案覆盖了一些方法（例如 kind()）。

exception BadValueError()
无法为属性分配值，因为该值对于该属性类型无效。
exception ConfigurationError()
属性配置错误。
exception DuplicatePropertyError()
Model 定义有多个名称相同的属性。
exception InternalError()
数据库服务发生内部错误。
exception KindError()
应用程序尝试将数据实体与不匹配该实体的 Model 类配合使用。
exception NotSavedError()
执行了一个需要将对象保存（放置）到数据库中的操作，但对象未保存。
exception PropertyError()
引用的模型属性在数据对象上不存在。
exception ReservedWordError()
Model 定义了其名称禁止使用的属性。请参阅禁止使用的属性名称。
exception Rollback()
表示事务中的函数想回滚事务而不是提交事务。事务中未捕捉的异常将导致该事务回滚。当某个函数要回滚且没有其他异常适用时，则该 exception 类可提供方便。
exception TransactionFailedError()
即使在重新尝试后也无法提交事务或数据库操作。这通常由高冲突率引起：许多其他应用程序实例同时更新数据，且该实例无法在固定的重新尝试次数内提交其事务。请参阅事务。

google.appengine.runtime.apiproxy_errors 包提供以下 exception 类：

exception CapabilityDisabledError()

表示未执行数据库 API 调用，因为该特定数据库功能不可用。

图像 API

许多网络应用程序需要处理图像，例如照片或用户头像。Google App Engine 提供了图像处理服务，以便在网络请求期间执行简单的图像转换。

本参考包括以下小节：

- 概览
- 使用图像 API
- 参考
  - Image
  - 函数
  - 异常

概述

通过使用 App Engine 图像服务，您的应用程序可以使用与 Picasa 网络相册相同的可扩展基础架构来控制图像。采用该 API，您可以对 JPEG、PNG、GIF（包括动画）、BMP、TIFF 和 ICO 格式的图像进行缩放、裁剪、旋转和翻转，并使用修正照片的自动算法调整它们的对比度和颜色级别。

可以在图像上执行转换以生成缩略图，也可以在应用程序管理课程中执行其他典型操作。

当前可用的转换有：

缩放

以相同的纵横比缩放图像。如果同时指定了宽度和高度，当调整照片大小时会使用两个值中限制性较强的一个。

旋转

沿顺时针方向按指定度数旋转图像。

水平翻转

水平翻转图像。

垂直翻转

垂直翻转图像。

裁剪

根据您传递给函数的边界框裁剪图像。

手气不错。

[手气不错] 转换功能可增强图像的暗度和亮度，并可将颜色和对比度调整到最佳水平。

注意：为了能在您的本地环境中使用图像 API，您必须先下载和安装 PIL（Python 映像库）。PIL 在 App Engine 中不可用，它只用作您本地环境中的图像 API 的存根。App Engine 上只能使用图像 API 中提供的转换。

```
from google.appengine.api import images

from google.appengine.ext import db
from google.appengine.ext import webapp

class Photo(db.Model):
  title = db.StringProperty()
  full_size_image = db.BlobProperty()
```

```
class Thumbnailer(webapp.RequestHandler):
  def get(self):
    if self.request.get("id"):
        photo= Photo.get_by_id(self.request.get("id"))

      if photo:
          img= images.Image(photo.full_size_image)
          img.resize(width=80, height=100)
          img.im_feeling_lucky()
          thumbnail= img.execute_transforms(output_encoding=images.JPEG)

        self.response.headers['Content-Type'] = 'image/jpeg'
        self.response.out.write(thumbnail)
        return

    # Either "id" wasn't provided, or there was no image with that ID
    # in the datastore.
    self.error(404)
```

使用图像 API

简介

图像 API 使您能够对图像执行一些常见的变换操作，比如调整大小、旋转以及调整颜色和对比度。通常，是对用户上传的图像或照片执行这些变换。本文档介绍了上传、变 换、存储以及动态提供图像的过程。我们将使用《使用入门指南》中的留言簿示例并对其进行修改以使用户可上传带有自己问候语的头像。

- 创建图像属性
- 上传用户图像
- 变换图像
- 动态提供图像

创建图像属性

我们需要做的第一件事情就是更新留言簿示例中的模型，以将上传的图像存储为二进制大对象。

```
class Greeting(db.Model):
  author= db.UserProperty()
  content= db.StringProperty(multiline=True)
  avatar= db.BlobProperty()
  date= db.DateTimeProperty(auto_now_add=True)
```

上传用户图像

下一步我们将修改表单以添加其他字段，以使用户可从其计算机中选择要上传的文件。我们还必须向表单标签添加 enctype 属性以指明我们要使其成为一个多部分表单帖子。

```
self.response.out.write("""
            <form action="/sign" enctype="multipart/form-data" method="post">
              <div><label>Message:</label></div>
              <div><textarea name="content" rows="3" cols="60"></textarea></div>
              <div><label>Avatar:</label></div>
              <div><input type="file" name="img"/></div>
              <div><input type="submit" value="Sign Guestbook"></div>
            </form>
          </body>
        </html>""")
```

这应当提供给我们一个包含两个字段的简单表单。

此时，我们还希望更新留言簿处理程序，以从表单帖子中获取图像数据，并将其作为 Blob 存储于数据库中。

```
class Guestbook(webapp.RequestHandler):
  def post(self):
      greeting= Greeting()
    if users.get_current_user():
        greeting.author= users.get_current_user()
      greeting.content= self.request.get("content")
    avatar= self.request.get("img")
      greeting.avatar= db.Blob(avatar)
      greeting.put()
    self.redirect('/')
```

变换图像

图像 API 可让您用几种不同的方式变换图像。

缩放

您可在保持纵横比不变的同时缩放图像。

旋转

您每次可以旋转图像 90 度。

水平翻转

您可以水平翻转图像。

垂直翻转

您可以垂直翻转图像。

裁剪

您可以使用指定的边框裁剪图像。

手气不错

[手气不错] 变换功能可增强图像的暗度和亮度，并可将颜色和对比度调整到最佳水平。

对于留言簿应用程序，我们将创建 32x32 大小的头像。首先我们必须导入 google.appengine.api.images 模块。然后我们需要调用 resize 函数并传入图像数据。

```
from google.appengine.apiimport images

class Guestbook(webapp.RequestHandler):
  def post(self):
      greeting= Greeting()
     if users.get_current_user():
          greeting.author= users.get_current_user()
      greeting.content= self.request.get("content")
     avatar= images.resize(self.request.get("img"), 32, 32)
      greeting.avatar= db.Blob(avatar)
```

```
    greeting.put()
  self.redirect('/')
```

动态提供图像

最后，我们需要创建一个图像处理程序，该程序将在 /img 路径之外动态提供这些图像。我们还将更新 HTML 以引入这些动态提供的图像。

```
class Image (webapp.RequestHandler):
 def get(self):
    greeting= db.get(self.request.get("img_id"))
  if greeting.avatar:
    self.response.headers['Content-Type'] = "image/png"
    self.response.out.write(greeting.avatar)
  else:
    self.error(404)
```

在图像处理程序中，我们从请求中获取 img_id。我们将需要更新留言簿的 HTML 以将问候语的键传递到该图像处理程序

```
self.response.out.write("<div><img src='img?img_id=%s'></img>" %
                                        greeting.key())
self.response.out.write(' %s</div>' %
                                        cgi.escape(greeting.content))
```

现在我们已完成修改的留言簿应用程序：

```
import cgi
import datetime
import logging

from google.appengine.ext import db
from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.api import images

logging.getLogger().setLevel(logging.DEBUG)


class Greeting(db.Model):
  author= db.UserProperty()
  content= db.StringProperty(multiline=True)
  avatar= db.BlobProperty()
```

```
    date= db.DateTimeProperty(auto_now_add=True)

class MainPage(webapp.RequestHandler):
  def get(self):
      self.response.out.write('<html><body>')
        query_str= "SELECT * FROM Greeting ORDER BY date DESC LIMIT 10"
        greetings= db.GqlQuery (query_str)

        for greetingin greetings:
          if greeting.author:
              self.response.out.write('<b>%s</b> wrote:' % greeting.author.nickname())
          else:
              self.response.out.write('An anonymous person wrote:')
          self.response.out.write("<div><img src='img?img_id=%s'></img>" %
                                              greeting.key())
          self.response.out.write(' %s</div>' %
                                              cgi.escape(greeting.content))

      self.response.out.write("""
                <form action="/sign" enctype="multipart/form-data" method="post">
                  <div><label>Message:</label></div>
                  <div><textarea name="content" rows="3" cols="60"></textarea></div>
                  <div><label>Avatar:</label></div>
                  <div><input type="file" name="img"/></div>
                  <div><input type="submit" value="Sign Guestbook"></div>
                </form>
              </body>
            </html>""")

class Image (webapp.RequestHandler):
  def get(self):
        greeting= db.get(self.request.get("img_id"))
      if greeting.avatar:
          self.response.headers['Content-Type'] = "image/png"
          self.response.out.write(greeting.avatar)
      else:
          self.response.out.write("No image")

class Guestbook(webapp.RequestHandler):
  def post(self):
        greeting= Greeting()
      if users.get_current_user():
            greeting.author= users.get_current_user()
        greeting.content= self.request.get("content")
```

```
        avatar= images.resize(self.request.get("img"), 32, 32)
        greeting.avatar= db.Blob(avatar)
        greeting.put()
      self.redirect('/')


application= webapp.WSGIApplication([
  ('/', MainPage),
  ('/img', Image),
  ('/sign', Guestbook)
], debug=True)


def main():
    run_wsgi_app(application)


if __name__== '__main__':
    main()
```

Image 类

Image 类的实例代表可以对其应用多个转换的单个图像。实例上的方法设置转换，这些转换在调用 execute_transforms() 方法时一次性全部执行。

Image 由 google.appengine.api.images 模块提供。

简介

Image 类用于封装图像信息和该图像的转换。

在 Image 对象上调用了一个或多个转换后，您可以用 execute_transforms() 方法执行转换。

注意：每个转换都以请求的顺序应用，并且针对每个图像的每次 execute_transforms() 调用只能调用一次。

构造函数

class Image(image_data)

要转换的图像。

参数：

image_data
: 字节字符串格式的图像数据 (str)。可以采用 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式对图像数据进行编码。

实例方法

Image 实例有以下方法：

resize(width=0, height=0)

缩放图像，缩小或放大到指定宽度和高度。

参数：

width
: 所需的宽度，以像素数量表示。必须为 int 或 long。

height
: 所需的长度，以像素数量表示。必须为 int 或 long。

crop(left_x, top_y, right_x, bottom_y)

将图像裁剪到指定边界框。方法以相同的格式返回转换的图像。

将边界框的左侧、顶部、右侧和底部指定为成比例的距离。边界框的坐标被确定为 left_x * width、top_y * height、right_x * width 和 bottom_y * height。这可让您指定与图像的最终宽度和高度无关的边界框，图像的最终宽度和高度可能与缩放操作同时更改。

参数：

left_x
: 边界框的左边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像宽度的比例。

top_y
: 边界框的上边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像高度的比例。

right_x
: 边界框的右边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像宽度的比例。

bottom_y

边界框的下边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像高度的比例。

rotate(degrees)

旋转图像。旋转量必须是 90 度的倍数。

旋转沿顺时针方向执行。旋转 90 度会使图像的上边缘成为右边缘。

参数：

degrees
旋转图像的量，采用度数的形式，是 90 度的倍数。必须为 int 或 long。

horizontal_flip()

水平翻转图像。左边缘变成右边缘，或者反之。

vertical_flip()

垂直翻转图像。上边缘变成下边缘，或者反之。

im_feeling_lucky()

根据改善照片的算法调整图像的对比度和颜色级别。这与 Google Picasa 的 [手气不错] 功能类似。方法以相同的格式返回转换的图像。

注意：当在 SDK 中本地使用 im_feeling_lucky() 方法时，该方法是 no-op，因为 PIL 上没有等效方法。

execute_transforms(output_encoding=images.PNG)

通过以上方法执行 Image 实例的所有转换集并返回结果。

参数：

output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

返回值是结果图像，形式是以请求的格式编码的字节字符串。

函数

google.appengine.api.images 包提供以下函数：

resize(image_data, width=0, height=0, output_encoding=images.PNG)

缩放图像，缩小或放大到指定宽度和高度。函数采用图像数据进行缩放，并以相同的格式返回转换后的图像。

参数：

image_data
要缩放的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
width
所需的宽度，以像素数量表示。必须为 int 或 long。
height
所需的高度，以像素数量表示。必须为 int 或 long。
output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

crop(image_data, left_x, top_y, right_x, bottom_y), output_encoding=images.PNG)

将图像裁剪到指定边界框。函数采用图像数据进行裁剪，并以相同的格式返回转换后的图像。

将边界框的左侧、顶部、右侧和底部指定为一定比例的距离。将边界框的坐标确定为 left_x * width、top_y * height、right_x * width 和 bottom_y * height。这可让您指定与图像的最终宽度和高度无关的边界框，图像的最终宽度和高度可能与缩放操作同时更改。

参数：

image_data
要裁剪的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
left_x
边界框的左边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像宽度的比例。
top_y
边界框的上边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像高度的比例。
right_x
边界框的右边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像宽度的比例。
bottom_y
边界框的下边界，采用指定为 float 值从 0.0 到 1.0（包括 0.0 和 1.0）的图像高度的比例。
output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

rotate(image_data, degrees, output_encoding=images.PNG)

旋转图像。旋转量必须是 90 度的倍数。函数采用图像数据进行旋转，并以相同的格式返回转换后的图像。

旋转沿顺时针方向执行。旋转 90 度会使图像的上边缘成为右边缘。

参数：

image_data
要旋转的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
degrees
旋转图像的量，以度数表示，是 90 度的倍数。
output_encoding

转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

horizontal_flip(image_data, output_encoding=images.PNG)

水平翻转图像。左边缘变成右边缘，或者反之。函数采用图像数据进行翻转，并以相同的格式返回转换后的图像。

参数：

image_data
要翻转的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

vertical_flip(image_data, output_encoding=images.PNG)

垂直翻转图像。上边缘变成下边缘，或者反之。函数采用图像数据进行翻转，并以相同的格式返回转换后的图像。

参数：

image_data
要翻转的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

im_feeling_lucky(image_data, output_encoding=images.PNG)

根据算法调整图像的对比度和颜色级别以改进图像。这与 Google Picasa 的 [手气不错] 功能类似。函数采用图像数据进行调整，并以相同的格式返回转换后的图像。

参数：

image_data
要调整的图像，是 JPEG、PNG、GIF（包括动画）、BMP、TIFF 或 ICO 格式的字节字符串 (str)。
output_encoding
转换的图像所需的格式。可以是 images.PNG 或 images.JPEG 格式。默认为 images.PNG。

注意：当在 SDK 中本地使用 im_feeling_lucky() 方法时，该方法是 no-op，因为 PIL 上没有等效方法。

异常

google.appengine.api.images 包提供以下 exception 类：

exception Error()

这是该包中所有异常的基类。

exception TransformationError()

尝试转换图像时发生错误。

exception BadRequestError()

转换参数无效。

exception NotImageError()

指定的图像数据不属于可以识别的图像格式。

exception BadImageError()

指定的图像数据被损坏。

exception LargeImageError()

指定的图像数据过大，无法处理。

邮件 API

对于应用程序而言，电子邮件是一种非常有用的与用户进行通信及通知其他人活动的方式。Google App Engine 包含了用于从您的应用程序发送电子邮件的电子邮件服务。

本参考包括以下小节：

概述

App Engine 提供从网络应用程序发送电子邮件的服务。

从应用程序发送的邮件的发件人地址可以是注册管理员的电子邮件地址，或当前登录用户（提出发送邮件请求的用户）的电子邮件地址。错误邮件和退回邮件会发送到发件人地址。发件人地址还会接收到该邮件的副本。

```
from google.appengine.api import mail

class ConfirmUserSignup(webapp.RequestHandler):
```

```
 def post(self):
      user_address= self.request.get("email_address")

    if not mail.is_email_valid(user_address):
      # prompt user to enter a valid address

    else:
        confirmation_url= createNewUserConfirmation(self.request)
        sender_address= "support@example.com"
        subject= "Confirm your registration"
        body= """
Thank you for creating an account!  Please confirm your email address by
clicking on the link below:

%s
""" % confirmation_url

        mail.send_mail(sender_address, user_address, subject, body)
```

发送邮件

邮件 API 提供两种方式来发送电子邮件：mail.send_mail() 函数以及 EmailMessage 类。

mail.send_mail() 函数将电子邮件字段（包括发件人、收件人、主题以及电子邮件正文）用作参数。

```
from google.appengine.api import mail

mail.send_mail(sender="support@example.com",
                      to="Albert Johnson <Albert.Johnson@example.com>",
                      subject="Your account has been approved",
                      body="""
Dear Albert:

Your example.com account has been approved.  You can now visit
http://www.example.com/ and sign in using your Google Account to
access new features.

Please let us know if you have any questions.

The example.com Team
""")
```

EmailMessage 类提供了与使用对象相同的功能。电子邮件字段可传递给 EmailMessage 构造函数，然后使用实例属性进行更新。send() 方法会发送由实例属性表示的电子邮件。应用程序可通过修改属性并再次调用 send() 方法重复

使用 EmailMessage 实例。

```
from google.appengine.api import mail

message = mail.EmailMessage(sender="support@example.com",
                            subject="Your account has been approved")

message.to = "Albert Johnson <Albert.Johnson@example.com>"
message.body = """
Dear Albert:

Your example.com account has been approved.  You can now visit
http://www.example.com/ and sign in using your Google Account to
access new features.

Please let us know if you have any questions.

The example.com Team
""")

message.send()
```

有关可能的电子邮件字段的详细信息，请参阅电子邮件字段。

发送是异步的：mail.send_mail() 函数和 EmailMessage send() 方法可将邮件数据传送到邮件服务，然后再返回。邮件服务会把邮件加入队列，然后尝试发送该邮件，如果目标邮件服务器不可用，则可能重新尝试发送该邮件。错误邮件和退回邮件将发送至该电子邮件的发件人地址。

发件人地址可以是该应用程序的注册管理员的电子邮件地址，也可以是当前登录用户（发出发送邮件请求的用户）的电子邮件地址。这里有一个从当前登录用户发送邮件的示例，如果用户未登录，可使用 login_required 注释将该用户重定向至登录页面：

```
from google.appengine.api import mail
from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import login_required

class InviteFriendHandler(webapp.RequestHandler):
  @login_required
  def post(self):
      to_addr = self.request.get("friend_email")
    if not mail.is_email_valid(to_addr):
        # Return an error message...
        pass
```

```
    message= mail.EmailMessage()
    message.sender= users.get_current_user().email()
    message.to= to_addr
    message.body= """
    I've invited you to Example.com!

    To accept this invitation, click the following link,
    or copy and paste the URL into your browser's address
    bar:

    %s
    """ % generate_invite_link(to_addr)

    message.send()
```

如果使用命令行选项将开发网络服务器配置为可发送电子邮件，那么，该开发网络服务器就能发送电子邮件。开发网络服务器可以使用 SMTP 服务器或 Sendmail 应用程序（如果有的话）。当您的应用程序在 App Engine 上运行时，该应用程序将使用 App Engine 邮件服务发送电子邮件。有关详细信息，请参阅开发网络服务器。

附件

电子邮件可以包含文件附件。电子邮件的 [attachments] 字段接受二值元组列表（每个附件对应一个元组）。元组的第一个元素是用于邮件中的附件的文件名，第二个元素是数据（采用字节字符串形式）。

```
from google.appengine.api import mail
from google.appengine.ext import db

class DocFile(db.Model):
  doc_name= StringProperty()
  doc_file= BlobProperty()

q= db.GqlQuery("SELECT * FROM DocFile WHERE doc_name = :1", myname)
results= q.fetch(1)
if results:
  doc= results[0]
  mail.send_mail(sender="support@example.com",
                 to="Albert Johnson <Albert.Johnson@example.com>",
                 subject="The doc you requested",
                 body="""
Attached is the document file you requested.

The example.com Team
""",
                 attachments=[(doc.doc_name, doc.doc_file)])
```

只允许某些类型的文件作为附件。附件的文件名则说明该文件的类型。

允许的附件类型

为了安全起见，附加到电子邮件的文件必须是以下允许的文件类型，并且文件名必须以与文件类型对应的扩展名结尾。附件以文件扩展名决定的 MIME 类型发送。

以下是允许作为电子邮件文件附件的 MIME 类型及其对应的文件扩展名列表。

MIME 类型	文件扩展名
图像/x-ms-bmp	bmp
文本/css	css
文本/逗号分隔值	csv
图像/gif	gif
文本/html	htm、html
图像/jpeg	jpeg、jpg、jpe
应用程序/pdf	pdf
图像/png	png
应用程序/rss+xml	rss
文本/plain	text、txt、asc、diff、pot
图像/tiff	tiff、tif
图像/vnd.wap.wbmp	wbmp

EmailMessage 类

EmailMessage 类的实例代表电子邮件，并包含使用 App Engine 邮件服务发送邮件的方法。

EmailMessage 由 google.appengine.api.mail 模块提供。

简介

EmailMessage 实例代表要使用 App Engine 邮件服务发送的电子邮件。电子邮件具有一组字段，这些字段可以使用构造函数进行初始化，并使用实例的属性进行调整。

有关 EmailMessage 的字段的列表，请参阅电子邮件字段。

如果设置了所有相应的字段，send() 方法会使用字段的当前值发送电子邮件。可以重复使用 EmailMessage 实例以便用相似的字段值发送多封邮件。

构造函数

class EmailMessage(**kw)

将通过邮件 API 发送的电子邮件。

邮件的字段可以使用传递到构造函数的关键字参数进行初始化。有关可能字段的列表，请参阅电子邮件字段。

字段还可以在构造之后使用实例的属性设置，或通过将关键字参数传递到 initialize() 方法来设置。

实例方法

EmailMessage 实例有以下方法：

check_initialized()

检查 EmailMessage 是否进行了正确的初始化以便发送。如果邮件没有正确地初始化，该方法会抛出与其找到的第一个问题对应的错误。如果邮件已准备好发送，则会返回而不抛出错误。

initialize(**kw)
: 使用关键字参数设置电子邮件的字段。有关可能字段的列表，请参阅电子邮件字段。

is_initialized()
: 如果 EmailMessage 进行了正确的初始化以便发送，则返回 True。这会与 check_initialized() 一样执行相同的检查，但是不抛出错误。

send()
: 发送电子邮件。

邮件字段

电子邮件有若干个信息字段。字段可以设置为 EmailMessage 构造函数的关键字参数、initialize() 方法的关键字参数或 EmailMessage 实例的属性。您可以通过调用 send_mail() 函数并将字段设置为关键字参数，来创建和发送单封电子邮件。

收件人电子邮件地址可以只是电子邮件地址 (Albert.Johnson@example.com)，或者是固定格式的姓名加电子邮件地址，例如 Albert Johnson <Albert.Johnson@example.com>。发件人地址只能是电子邮件地址。

以下是电子邮件中可能包含的字段：

sender
: 发件人的电子邮件地址（[From] 地址）。该地址必须是应用程序的注册管理员的电子邮件地址，或当前登录的用户的电子邮件地址。可以使用管理控制台将管理员添加到应用程序。当前用户的电子邮件地址可以通过用户 API

确定。

to

显示在邮件信头中 [To:] 行中的收件人的电子邮件地址（字符串）或电子邮件地址列表。

cc

显示在邮件信头中 [Cc:] 行中的收件人的电子邮件地址（字符串）或电子邮件地址列表。

bcc

收件人的电子邮件地址（字符串）或要接收邮件的电子邮件地址列表，但是不显示在邮件信头中（[密件抄送]）。

reply_to

收件人应回复至的电子邮件地址（不是 [sender] 地址），[Reply-To:] 字段。

subject

邮件的主题（[Subject:] 行）。

body

邮件的纯文本正文内容。

html

HTML 版本的正文内容，适合于喜欢 HTML 电子邮件的收件人。

attachments

邮件的文件附件，是一个二值元组列表（每个附件对应一个元组）。每个元组都包含两个元素，第一个为文件名，第二个为文件内容。

附件文件必须是允许的文件类型，而且文件名必须以与类型对应的扩展名结尾。有关允许的文件类型和文件扩展名的列表，请参阅允许的附件类型。

函数

google.appengine.api.mail 包提供以下函数：

check_email_valid(email_address, field)

检查电子邮件地址是否有效，如果无效则会抛出 InvalidEmailError 异常。field 是包含该地址的字段的名称，用于错误消息。

invalid_email_reason(email_address, field)

返回有关指定电子邮件地址无效的原因的字符串说明，如果该地址有效，则返回 None。field 是包含该地址的字段的名称，用于错误消息。

is_email_valid(email_address)

如果 email_address 是有效的电子邮件地址，则返回 True。这会执行与 check_email_valid 相同的检查，但是不抛出异常。

send_mail(sender, to, subject, body, **kw)

创建并发送一封电子邮件。sender、to、subject 和 body 是邮件的必填字段。其他字段可以指定为关键字参数。有

关可能字段的列表，请参阅电子邮件字段。

send_mail_to_admins(sender, subject, body, **kw)

创建并发送一封电子邮件通知应用程序的所有管理员。sender、subject 和 body 是邮件的必填字段。其他字段可以指定为关键字参数。有关可能字段的列表，请参阅电子邮件字段。

异常

google.appengine.api.mail 包提供以下 exception 类：

exception Error()

这是该包中所有异常的基类。

exception BadRequestError()

邮件服务以无效为理由拒绝了 EmailMessage，且没有其他错误适用。

exception InvalidSenderError()

EmailMessage 具有一个对于该应用程序无效的 sender。发件人必须是该应用程序注册管理员的电子邮件地址，或当前已登录的用户的地址。可以使用管理控制台将管理员添加到应用程序。当前用户的电子邮件地址可以通过用户 API 确定。

exception InvalidEmailError()

电子邮件地址无效。电子邮件地址字段仅接受有效的电子邮件地址，例如 sender 或 to。

exception InvalidAttachmentTypeError()

EmailMessage 至少有一个附件具有不允许使用的文件扩展名。请参阅允许的附件类型以查看允许使用的文件扩展名列表。

exception MissingRecipientsError()

EmailMessage 没有在以下任何字段设置收件人：to、cc、bcc。必须设置至少一个收件人才能发送邮件。

exception MissingSenderError()

EmailMessage 的 sender 字段缺少值。必须设置 sender 才能发送邮件。

exception MissingSubjectError()

EmailMessage 的 subject 字段缺少值。必须设置 subject 才能发送邮件。

exception MissingBodyError()

EmailMessage 的 body 字段缺少值。必须设置 body 才能发送邮件。

Memcache API

高性能可扩展网络应用程序通常在某些任务的稳定持久存储之前使用分布式内存数据缓存，或用分布式内存数据缓存来代替某些任务的稳定持久存储。Google App Engine 包含了有此用途的内存缓存服务。

本参考包括以下小节：

概述

Memcache 服务为您的应用程序提供了高性能的内存键值缓存，您可通过应用程序的多个实例访问该缓存。Memcache 对于那些不需要数据库的永久性功能和事务功能的数据很有用，例如临时数据或从数据库复制到缓存以进行高速访问的数据。Memcache API 与 Danga Interactive 开发的 Memcached 有类似的功能并兼容。

Memcache API 可通过以下方式让您提高应用程序的性能并减少数据库的负载：

- 显著地减少数据库查询的次数。
- 减少使用率非常高的页面的数据库配额的使用。
- 缓存操作量巨大的查询和操作的结果。
- 让使用临时计数器成为可能。

通过使用 Memcache API，您可以为应用程序中的数据创建一致的缓存。缓存可用于应用程序中的所有实例，而且数据只有通过内存压力（例如缓存中的数据过多）或开发人员设置的 缓存政策才能清除。可以在存储在缓存中的每个键-值对上设置缓存政策。您可以清除所有缓存或针对每份数据设置缓存过期时间。

```
from google.appengine.api import memcache

# Add a value if it doesn't exist in the cache, with a cache expiration of 1 hour.
memcache.add(key="weather_USA_98105", value="raining", time=3600)

# Set several values, overwriting any existing values for these keys.
memcache.set_multi({ "USA_98105": "raining",
                     "USA_94105": "foggy",
                     "USA_94043": "sunny" },
```

```
                    key_prefix="weather_", time=3600)

# Atomically increment an integer value.
memcache.set(key="counter", 0)
memcache.incr("counter")
memcache.incr("counter")
memcache.incr("counter")
```

使用 Memcache

Memcache 是一个高性能的分布式内存对象缓存系统，主要用于快速访问缓存的数据库查询结果。

Memcache 模式

Memcache 通常与以下模式配合使用：

- 应用程序将接收来自用户或应用程序的查询。
- 应用程序检查满足该查询所需的数据是否在 Memcache 中。
  如果数据在 Memcache 中，应用程序将使用该数据。
  如果数据不在 Memcache 中，应用程序将查询数据库并将结果存储在 Memcache 中以便将来提出请求。

以下伪代码表示一个典型的 Memcache 请求：

```
def get_data():
   data= memcache.get("key")
 if datais not None:
     return data
 else:
      data= self.query_for_data()
      memcache.add("key", data, 60)
    return data
```

修改 guestbook.py 以使用 Memcache

《使用入门指南》中的留言簿应用程序将基于每个请求查询数据库。您可以将留言簿应用程序修改为在查询数据库之前使用 Memcache。

首先，我们将导入 Memcache 模块，并创建在运行查询前检查 Memcache 的方法。

```
from google.appengine.api import memcache
```

```
def get_greetings(self):
  """get_greetings()

     Checks the cache to see if there are cached greetings.
   If not, call render_greetings and set the cache

   Returns:
      A string of HTML containing greetings.
   """
   greetings= memcache.get("greetings")
  if greetingsis not None:
     return greetings
  else:
       greetings= self.render_greetings()
     if not memcache.add("greetings", greetings, 10):
          logging.error("Memcache set failed.")
     return greetings
```

然后，我们将分出查询和页面 HTML 的创建。如果缓存未命中，我们将调用此方法来查询数据库并构建将存储在 Memcache 中的 HTML 字符串。

```
def render_greetings(self):
  """render_greetings()

     Queries the database for greetings, iterate through the
   results and create the HTML.

   Returns:
      A string of HTML containing greetings
   """
   results= db.GqlQuery("SELECT * "
                        "FROM Greeting "
                        "ORDER BY date DESC").fetch(10)
   output= StringIO.StringIO()
  for resultin results:
     if result.author:
          output.write("<b>%s</b> wrote:" % result.author.nickname())
     else:
          output.write("An anonymous person wrote:")
       output.write("<blockquote>%s</blockquote>" %
                    cgi.escape(result.content))
  return output.getvalue()
```

最后，我们将更新 MainPage 处理程序以调用 get_greetings() 方法并显示有关缓存命中或未命中的次数的一些统计

信息。

```
import cgi
import datetime
import logging
import StringIO

from google.appengine.ext import db
from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.api import memcache

logging.getLogger().setLevel(logging.DEBUG)


class Greeting(db.Model):
    author = db.UserProperty()
    content = db.StringProperty(multiline=True)
    date = db.DateTimeProperty(auto_now_add=True)


class MainPage(webapp.RequestHandler):
  def get(self):
      self.response.out.write("<html><body>")
      greetings = self.get_greetings()
       stats = memcache.get_stats()

       self.response.out.write("<b>Cache Hits:%s</b><br>" % stats['hits'])
      self.response.out.write("<b>Cache Misses:%s</b><br><br>" %
                                      stats['misses'])
      self.response.out.write(greetings)
      self.response.out.write("""
                <form action="/sign" method="post">
                  <div><textarea name="content" rows="3" cols="60"></textarea></div>
                  <div><input type="submit" value="Sign Guestbook"></div>
                </form>
              </body>
            </html>""")

  def get_greetings(self):
      """
            get_greetings()
            Checks the cache to see if there are cached greetings.
```

```
            If not, call render_greetings and set the cache

            Returns:
                A string of HTML containing greetings.
        """
        greetings= memcache.get("greetings")
        if greetingsis not None:
            return greetings
        else:
            greetings= self.render_greetings()
            if not memcache.add("greetings", greetings, 10):
                logging.error("Memcache set failed.")
            return greetings

    def render_greetings(self):
        """
            render_greetings()
            Queries the database for greetings, iterate through the
            results and create the HTML.

            Returns:
                A string of HTML containing greetings
        """
        results= db.GqlQuery("SELECT * "
                             "FROM Greeting "
                             "ORDER BY date DESC").fetch(10)
        output= StringIO.StringIO()
        for resultin results:
            if result.author:
                output.write("<b>%s</b> wrote:" % result.author.nickname())
            else:
                output.write("An anonymous person wrote:")
            output.write("<blockquote>%s</blockquote>" %
                         cgi.escape(result.content))
        return output.getvalue()

class Guestbook(webapp.RequestHandler):
    def post(self):
        greeting= Greeting()

        if users.get_current_user():
            greeting.author= users.get_current_user()

        greeting.content= self.request.get('content')
```

```
        greeting.put()
      self.redirect('/')


application= webapp.WSGIApplication([
  ('/', MainPage),
  ('/sign', Guestbook)
], debug=True)


def main():
    run_wsgi_app(application)


if __name__ == '__main__':
    main()
```

Client 类

Memcache API 提供一个基于类的接口，以便与其他 Memcache API 兼容。另请参阅提供相同功能的函数接口。

Client 类由 google.appengine.api.memcache 模块提供。

简介

作为开发人员，您要通过 Memcache 客户端对象调用所有 Memcache 操作。

有些方法为 no-op，而有些方法值则会被忽略以便维持与现有常用 Python Memcache 库在源级别上的兼容性。例如，Google App Engine 透明地处理分片，但仍允许任何采用 [key] 参数的方法，接受该键作为字符串或 (哈希值, 字符串) 格式的元组。通常该哈希值用于分片为 Memcache 实例，但如果采用 App Engine，则该值将被忽略。有关详细信息，请参阅 Memcached 兼容性。

构造函数

class Client()

与 Memcache 服务通信的客户端。

实例方法

Client 实例有以下方法：

set(key, value, time=0, min_compress_len=0)

设置键的值，与先前缓存中的内容无关。

参数：

key
: 要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

value
: 要设置的值。

time
: 可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

min_compress_len
: 为了兼容性而忽略的选项。

如果设置的话，则返回值为 True，如果错误，则返回值为 False。

set_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时设置多个键的值。减少连续执行多个请求时的网络延迟。

参数：

mapping
: 键到值的参照表。

time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix
用于预置所有键的前缀。

min_compress_len
为了兼容性而忽略的选项。

返回值是未设置值的键的列表。全部成功时，该列表应为空。

get(key)

在 Memcache 中查找一个键。

参数：

key
要在 Memcache 中查找的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

如果在 Memcache 中找到键，则返回值为该键的值，否则为 None。

get_multi(keys, key_prefix='')

通过一个操作从 Memcache 中查找多个键。

参数：

keys
要查找的键的列表。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

key_prefix
与服务器通讯期间用于预置所有键的前缀，不包含在返回的参照表中。

返回的值是曾存在于 Memcache 中的键和值的参照表。即使指定了 key_prefix，在返回的参照表中，键也不会包含 key_prefix。

delete(key, seconds=0)

从 Memcache 删除键。

参数：

key
要删除的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

seconds
使删除的项目对 [添加] 操作 [锁定] 的可选秒数。值可以是从当前时间开始的增量（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认值为 0，表示可以立即添加项目。无论是否有此选项，[设置] 操作都始终有效。浮点值将四舍五入为最接近的整秒。

网 络故障时返回值为 0 (DELETE_NETWORK_FAILURE)，如果服务器尝试删除项目但其实并没有项目，则返回值为 1 (DELETE_ITEM_MISSING)；如果确实删除了项目，则返回值为 2 (DELETE_SUCCESSFUL)。这可以用作布尔值，其中网络故障是唯一的不良状况。

delete_multi(keys, seconds=0, key_prefix='')

同时删除多个键。

参数：

keys
要删除的键的列表。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。
seconds
使删除的项目对 [添加] 操作 [锁定] 的可选秒数。值可以是从当前时间开始的增量（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认值为 0，表示可以立即添加项目。无论是否有此选项，[设置] 操作都始终有效。浮点值将四舍五入为最接近的整秒。
key_prefix
当向 Memcache 发送指定的键时要添加到所有键的前缀。请参阅 get_multi() 和 set_multi() 的文档。

如果所有操作成功完成，则返回值为 True。如果一个或多个操作没有完成，则返回 False。

add(key, value, time=0, min_compress_len=0)

仅当项目尚未在 Memcache 中时，才设置键的值。

参数：

key
要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。
value
要设置的值。
time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。
min_compress_len
为了兼容性而忽略的选项。

如果已添加，则返回值为 True，如果错误，则返回值为 False。

add_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时添加多个值，对于已经在 Memcache 中的键没有影响。

参数：

mapping
: 键到值的映射。

time
: 可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix
: 当向 Memcache 发送指定的键时要添加到所有键的前缀。请参阅 get_multi()。

min_compress_len
: 为了兼容性而忽略的选项。

返回值是键的列表，这些键的值未设置，因为它们已在 Memcache 中设置；返回值也可能是空列表。

replace(key, value, time=0, min_compress_len=0)

替换键的值，如果项目尚未在 Memcache 中，则会失败。

参数：

key
: 要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

value
: 要设置的值。

time
: 可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

min_compress_len
: 为了兼容性而忽略的选项。

如果替换掉，则返回值为 True。如果出现错误或者缓存未命中，则返回值为 False。

replace_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时替换多个值，对于不在 Memcache 中的键没有影响。

参数：

mapping
: 键到值的映射。

time

可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix

当向 Memcache 发送指定的键时要添加到所有键的前缀。请参阅 get_multi()。

min_compress_len

为了兼容性而忽略的选项。

返回值是键的列表，这些键的值未设置，因为它们未在 Memcache 中设置；返回值也可能是空列表。

incr(key, delta=1)

自动增加键的值。在内部，值是无符号 64 位整数。Memcache 不检查 64 位溢出。值如果过大则会换行。

键必须已存在于缓存中才能增加值。要初始化计数器，请使用 set() 将其设置为初始值，如 ASCII 十进制整数。将来，在增加以后，使用 get() 获取的键将仍是 ASCII 十进制值。

参数：

key

要增加的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

delta

作为键的增加量的非负整数值（整型或长整型），默认为 1。

返回值为新的长整型整数值，或者是 None（如果键不在缓存中或因任何其他原因无法增加）。

decr(key, delta=1)

自动减少键的值。在内部，值是无符号 64 位整数。Memcache 不检查 64 位溢出。值如果过大则会换行。

键必须已存在于缓存中才能减少值。要初始化计数器，请使用 set() 将其设置为初始值，如 ASCII 十进制整数。将来，在减少以后，使用 get() 获取的键将仍是 ASCII 十进制值。

参数：

key

要减少的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

delta

作为键的减少量的非负整数值（整型或长整型），默认为 1。

返回值为新的长整型整数值，或者是 None（如果键不在缓存中或因任何其他原因无法减少）。

flush_all()

删除 Memcache 中的所有内容。

如果成功，则返回值为 True，如果是 RPC 或服务器错误，则返回值为 False。

get_stats()

获取该应用程序的 Memcache 统计信息。所有这些统计信息都可以根据不同的临时条件而重设。它们会提供在被调用时可用的最佳信息。

返回值是将统计信息名称映射到相关值的参照表。统计信息及其相关意义：

hits：导致缓存命中的缓存 get 请求的次数。
misses：导致缓存未命中的缓存 get 请求的次数。
byte_hits：get 请求时转移的总字节。溢出时转为零。
items：缓存中键/值对的数量。
bytes：缓存中所有项目的总大小。
oldest_item_age：从访问缓存中最早的项目起过了多少秒。实际上，这表示新项目在不被访问的情况下在缓存中继续有效的时间。这并不是从项目创建开始后经过的时间量。

Memcached 兼容性

Client 类包含一些方法和参数以便与 Memcached API 兼容。由于这些方法和参数中有的不适用于 App Engine Memcache 服务，所以它们不起作用。

为了兼容性而提供，但对 App Engine Memcache API 不起作用的方法包括：

- disconnect_all()
- set_servers(...)

函数

google.appengine.api.memcache 包提供以下函数：

set(key, value, time=0, min_compress_len=0)

设置键的值，与先前缓存中的内容无关。

参数：

key
要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。
value
要设置的值。
time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。
min_compress_len

为了兼容性而忽略的选项。

如果设置的话，则返回值为 True，如果错误，则返回值为 False。

set_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时设置多个键的值。减少连续执行多个请求时的网络延迟。

参数：

mapping
: 键到值的参照表。

time
: 可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix
: 用于预置所有键的前缀。

min_compress_len
: 为了兼容性而忽略的选项。

返回值是未设置值的键的列表。全部成功时，该列表应为空。

get(key)

在 Memcache 中查找一个键。

参数：

key
: 要在 Memcache 中查找的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

如果在内存缓存中找到键，则返回值为该键的值，否则为 None。

get_multi(keys, key_prefix='')

通过一个操作从 Memcache 中查找多个键。这是建议的批量加载方式。

参数：

keys
: 要查找的键的列表。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

key_prefix
: 与服务器通讯期间用于预置所有键的前缀，不包含在返回的参照表中。

返回的值是曾存在于 Memcache 中的键和值的参照表。即使指定了 key_prefix，在返回的参照表中，键也不会包含 key_prefix。

delete(key, seconds=0)

从 Memcache 删除键。

参数：

key
要删除的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。
seconds
使删除的项目对 [添加] 操作 [锁定] 的可选秒数。值可以是从当前时间开始的增量（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认值为 0，表示可以立即添加项目。无论是否有此选项，[设置] 操作都始终有效。浮点值将四舍五入为最接近的整秒。

网 络故障时返回值为 0 (DELETE_NETWORK_FAILURE)，如果服务器尝试删除项目但其实并没有项目，则返回值为 1 (DELETE_ITEM_MISSING)；如果确实删除了项目，则返回值为 2 (DELETE_SUCCESSFUL)。这可以用作布尔值，其中网络故障是唯一的不良状况。

delete_multi(keys, seconds=0, key_prefix='')

同时删除多个键。

参数：

keys
要删除的键的列表。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。
seconds
使删除的项目对 [添加] 操作 [锁定] 的可选秒数。值可以是从当前时间开始的增量（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认值为 0，表示可以立即添加项目。无论是否有此选项，[设置] 操作都始终有效。浮点值将四舍五入为最接近的整秒。
key_prefix
当向 Memcache 发送特定键时添加到所有键的前缀。请参阅 get_multi() 和 set_multi() 的文档。

如果所有操作成功完成，则返回值为 True。如果一个或多个操作没有完成，则返回 False。

add(key, value, time=0, min_compress_len=0)

仅当项目尚未在 Memcache 中时，才设置键的值。

参数：

key

要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

value
要设置的值。

time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

min_compress_len
为了兼容性而忽略的选项。

如果已添加，则返回值为 True，如果错误，则返回值为 False。

add_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时添加多个值，对于已经在 Memcache 中的键没有影响。

参数：

mapping
键到值的映射。

time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix
当向 Memcache 发送指定的键时要添加到所有键的前缀。请参阅 get_multi()。

min_compress_len
为了兼容性而忽略的选项。

返回值是键的列表，这些键的值未设置，因为它们已在 Memcache 中设置；返回值也可能是空列表。

replace(key, value, time=0, min_compress_len=0)

替换键的值，如果项目尚未在 Memcache 中，则会失败。

参数：

key
要设置的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

value
要设置的值。

time
可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

min_compress_len

为了兼容性而忽略的选项。

如果替换，返回的值为 True。如果错误或者缓存未命中，返回值为 False。

replace_multi(mapping, time=0, key_prefix='', min_compress_len=0)

同时替换多个值，对于不在 Memcache 中的键没有影响。

参数：

mapping
: 键到值的映射。

time
: 可选的过期时间，可以是相对当前时间的秒数（最多 1 个月），也可以是绝对 Unix 时间戳时间。默认情况下，项目永不过期，虽然项目可能由于内存压力而被去除。浮点值将四舍五入为最接近的整秒。

key_prefix
: 当向 Memcache 发送指定的键时要添加到所有键的前缀。请参阅 get_multi()。

min_compress_len
: 为了兼容性而忽略的选项。

返回值是键的列表，这些键的值未设置，因为它们未在 Memcache 中设置；返回值也可能是空列表。

incr(key, delta=1)

自动增加键的值。在内部，值是无符号 64 位整数。Memcache 不检查 64 位溢出。值如果过大则会换行。

键必须已存在于缓存中才能增加值。要初始化计数器，请使用 set() 将其设置为初始值，如 ASCII 十进制整数。将来，在增加以后，使用 get() 获取的键将仍是 ASCII 十进制值。

参数：

key
: 要增加的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

delta
: 作为键的增加量的非负整数值（整型或长整型），默认为 1。

返回值为新的长整型整数值，或者是 None（如果键不在缓存中或因任何其他原因无法增加）。

decr(key, delta=1)

自动减少键的值。在内部，值是无符号 64 位整数。Memcache 不检查 64 位溢出。值如果过大则会换行。

键必须已存在于缓存中才能减少值。要初始化计数器，请使用 set() 将其设置为初始值，如 ASCII 十进制整数。将来，在减少以后，使用 get() 获取的键将仍是 ASCII 十进制值。

参数：

key
: 要减少的键。Key 可以是字符串或 (哈希值, 字符串) 格式的元组，其中哈希值（通常用于分片为 Memcache 实例）会被忽略，因为 Google App Engine 会透明地处理分片。

delta
: 作为键的减少量的非负整数值（整型或长整型），默认为 1。

返回值为新的长整型整数值，或者是 None（如果键不在缓存中或因任何其他原因无法减少）。

flush_all()

删除 Memcache 中的所有内容。

如果成功，则返回值为 True，如果是 RPC 或服务器错误，则返回值为 False。

get_stats()

获取该应用程序的 Memcache 统计信息。所有这些统计信息都可以根据不同的临时条件而重设。它们会提供在被调用时可用的最佳信息。

返回值是将统计信息名称映射到相关值的参照表。统计信息及其相关意义：

hits：导致缓存命中的缓存 get 请求的次数。
misses：导致缓存未命中的缓存 get 请求的次数。
byte_hits：get 请求时转移的总字节。溢出时转为零。
items：缓存中键/值对的数量。
bytes：缓存中所有项目的总大小。
oldest_item_age：从访问缓存中最早的项目起过了多少秒。实际上，这表示新项目在不被访问的情况下在缓存中继续有效的时间。这并不是从项目创建开始后经过的时间量。

网址抓取 API

App Engine 应用程序可以使用 HTTP 请求通过互联网与其他主机进行通信。网址抓取 API 可以使用 HTTP 和 HTTPS 网址检索数据。

本参考包括以下小节：

概述

App Engine 应用程序可以与其他应用程序进行通信或通过抓取网址访问网络上的其他资源。这对与网络服务器通讯或检索 RSS 供稿数据非常有用。

urlfetch.fetch() 函数执行 HTTP 请求。

```
from google.appengine.api import urlfetch

url = "http://www.google.com/"
result = urlfetch.fetch(url)
if result.status_code == 200:
    doSomethingWithResult(result.content)
```

fetch() 支持五种 HTTP 方法：GET、POST、HEAD、PUT 和 DELETE。请求可以包含 HTTP 标头和 POST 或 PUT 请求的正文内容。例如，要使用 POST 操作向网络表单处理程序提交数据：

```
import urllib

form_fields = {
  "first_name": "Albert",
  "last_name": "Johnson",
  "email_address": "Albert.Johnson@example.com"
}
form_data = urllib.urlencode(form_fields)
result = urlfetch.fetch(url=url,
                        payload=form_data,
                        method=urlfetch.POST,
                        headers={'Content-Type': 'application/x-www-form-urlencoded'})
```

urlfetch 需知

仅限于在访问端口 80 (http) 和端口 443 (https) 中进行网址抓取。不支持其他端口或网址方案。

其他网址抓取操作是同步的：在远程服务器响应之前 fetch() 函数不会返回。

由于应用程序必须在若干秒内响应用户的请求，对较慢的远程服务器的网址抓取操作可能会导致应用程序向用户返回服务器错误。目前没有方法指定网址抓取 操作的时间限制。有一种控制较慢的远程服务器的用户体验的方式是：使用浏览器 JavaScript 在您的应用程序（该应用程序执行需要网址抓取的操作）上调用单独的处理程序，然后在处理程序失败的情况下向用户报告错误。

App Engine 使用与 HTTP/1.1 兼容的代理来抓取结果。代理可以提出 HTTPS 请求，但是无法验证与其联系的主机。代理接受所有 SSL 证书，包括自签名证书。

fetch() 可跟随 HTTP 重定向最多 5 次，然后返回最终资源。您可以用参数告知它不跟随重定向。

请求处理程序无法抓取自己的请求的网址。这可以避免意外的无限循环，出现无线循环情况时应用程序会一直调用

自己。

当发送 HTTP POST 请求时，如果未明确设置 Content-Type 标头，则标头会被设置为 x-www-form-urlencoded。这是网络表单使用的内容类型。

fetch 函数

google.appengine.api.urlfetch 包提供以下函数：

fetch(url, payload=None, ethod=GET,headers={}, allow_ runcated=False, follow_redirect=True)

在 url 中给出的网址抓取文档并返回包括响应详细信息的对象。有关返回值的详细信息，请参阅 Response 对象。

参数：

url
: 一个 http 或 https 网址。如果网址无效，将抛出 InvalidURLError。

payload
: POST 或 PUT 请求的正文内容。

method
: 用于请求的 HTTP 方法。可接受的值包括 GET、POST、HEAD、PUT 和 DELETE。这些值是由包提供的常量。

headers
: 请求中包含的 HTTP 标头的集合，作为名称和值的映射。出于安全目的，有的 HTTP 标头不能由应用程序修改。请参阅禁止使用的 HTTP 标头。

allow_truncated
: 如果为 False 且响应数据超过允许的响应大小上限，则将抛出 ResponseTooLargeError 异常。如果为 True，则不抛出异常，并且响应的 content 被截断为大小上限，Response 对象的 content_was_truncated 属性设置为 True。

follow_redirects
: 如果为 True，HTTP 重定向的响应后有最多 5 个连续重定向。响应数据来自最终位置，就像数据是用于请求的位置。如果为 False，则重定向后没有其他操作，且重定向响应会直接返回到应用程序，其中包括说明重定向的标头信息。

抓取操作是同步的。在服务器响应之前，不会返回 fetch()。较慢的远程服务器可能导致您的应用程序自身的请求超时。

注意：网址抓取无法验证 https 请求的服务器，因为没有证书信任链。代理程序将接受所有证书，包括自签名证书。

禁止使用的 HTTP 标头

出于安全目的，以下传出 HTTP 请求的 HTTP 标头不能由应用程序修改。

- Content-Length

- Host
- Referer
- User-Agent
- Vary
- Via
- X-Forwarded-For

Response 对象

fetch() 函数返回的对象包含由网址的服务器返回的响应详细信息。该对象具有以下几个属性：

content
: 响应的正文内容。

content_was_truncated
: 如果 fetch() 的 allow_truncated 参数为 True，且响应超出了响应大小上限，则为 True。在这种情况下，content 属性会包含截断的响应。

status_code
: HTTP 状态代码。

headers
: HTTP 响应标头，作为从名称到值的映射。

异常

google.appengine.api.urlfetch 包提供以下 exception 类：

exception Error()

这是该包中所有异常的基类。

exception InvalidURLError()

请求的网址不是有效网址，或使用了不受支持的方法。仅支持 http 和 https 网址。

exception DownloadError()

检索数据时发生错误。

如果服务器返回 HTTP 错误代码，则不抛出该异常：在这种情况下，响应数据原样返回，包括错误代码。

exception ResponseTooLargeError()

响应数据超过允许的大小上限，且传递到 fetch() 的 allow_truncated 参数为 False。

用户 API

Google App Engine 功能与 Google 帐户集成：应用程序可以让用户使用他们的 Google 帐户登录，并知道谁在会话期间使用应用程序。Google 帐户可以让您的用户无需创建新帐户，因而他们可以更快地进入您的应用程序，而且您的应用程序可以个性化用户体验而无需管理自身的登录系统。

本参考包括以下小节：

概述

App Engine 应用程序可以使用 Google 帐户验证用户。应用程序可以将用户重定向到 Google 帐户页面以便登录、注册帐户或退出。当用户使用 Google 帐户登录到应用程序时，应用程序可以获得用户的电子邮件地址和昵称。应用程序还可以检测用户是否是该应用程序的管理员，便于实施应用程序的仅供管理员使用 的区域。

```
from google.appengine.api import users

class MyHandler(webapp.RequestHandler):
  def get(self):
      user = users.get_current_user()
    if user:
        greeting = ("Welcome, %s! (<a href=\"%s\">sign out</a>)" %
                    (user.nickname(), users.create_logout_url("/intl/zh-CN/")))
    else:
        greeting = ("<a href=\"%s\">Sign in or register</a>." %
                    users.create_login_url("/intl/zh-CN/"))

    self.response.out.write("<html><body>%s</body></html>" % greeting)
```

User 对象

User 类的实例代表一个用户。User 实例是唯一的，并且是可比较的。如果两个实例相同，那么这两个实例代表同一用户。

应用程序可通过调用 users.get_current_user() 函数访问当前用户的 User 实例。

```
from google.appengine.api import users

user = users.get_current_user()
if not user:
  # The user is not signed in.
else:
  print "Hello, %s!" % user.nickname()
```

也可从电子邮件地址中构造 User 实例。

```
user = users.User("Albert.Johnson@example.com")
```

如果使用与有效 Google 帐户不相符的电子邮件地址调用 User 构造函数，则会创建对象，但该对象不会与真正的 Google 帐户匹配。即使有人在存储了该对象后用指定的电子邮件地址创建了 Google 帐户，也将出现这种情况。当用一个不代表 Google 帐户的电子邮件地址创建了 User 值时，该 User 值将永远不会与代表真正用户的 User 值相匹配。

当在开发网络服务器下运行时，所有 User 对象在存储于（模拟）数据库中时都假定代表有效的 Google 帐户。

将 User 值与数据库配合使用

User 实例可以是数据库属性值。

```
class UserPrefs(db.Model):
  user = db.UserProperty()

user = users.get_current_user()
if user:
  q = db.GqlQuery("SELECT * FROM UserPrefs WHERE user = :1", user)
  userprefs = q.get()
```

注意：此时，用户 API 不为 Google 帐户提供永久性的唯一标识符。虽然一个电子邮件地址在某指定时刻是某个帐户所特有的，但用户可随时更改该帐户的电子邮件地址。对电子邮件地址的更改不会自动传播到数据库 User 值。在以后的版本中可实现唯一用户 ID 与电子邮件地址的更改传播。在这些功能中的任一功能得以实现之前，没有好方法可以在电子邮件地址发生更改后将当前用户与该用户的偏好数据相关联。

避免在数据库实体键名中使用用户的电子邮件地址。如果您的应用程序存储了用户偏好数据，请考虑使用 User 属性和查询来检索数据。在实现传播功能之后，此项技术将对电子邮件地址更改自动生效。这并不能防止意外创建多个使用偏好对象（该操作将需要一个可计算的键名和一个事务），但查询由于排序的原因始终会返回正确的结果。

登录网址

用户 API 提供用于构建到 Google 帐户的网址的函数，Google 帐户可允许用户登录或退出，然后重定向回您的应用程序。

users.create_login_url() 和 users.create_logout_url() 都采用应用程序的目标网址，并返回用于登录或退出的 Google 帐户网址，该网址重定向回后面指定的网址。

```
from google.appengine.api import users

class MyHandler(webapp.RequestHandler):
  def get(self):
      user = users.get_current_user()
    if user:
        greeting = ("Welcome, %s! (<a href=\"%s\">sign out</a>)" %
                    (user.nickname(), users.create_logout_url("/intl/zh-CN/")))
    else:
        greeting = ("<a href=\"%s\">Sign in or register</a>." %
                    users.create_login_url("/intl/zh-CN/"))

    self.response.out.write("<html><body>%s</body></html>" % greeting)
```

开发网络服务器可以使用自己的登录和退出工具模拟 Google 帐户。当您登录到开发网络服务器上的应用程序时，服务器会提示您提供用于会话的电子邮件地址。有关详细信息，请参阅开发网络服务器。

管理员用户

应用程序可以测试当前登录的用户是否为该应用程序的注册管理员。管理员是可以访问应用程序的管理控制台的用户。您可以使用管理控制台来管理哪些用户有管理员身份。

如果当前用户是应用程序的管理员，函数 users.is_current_user_admin() 将返回 True。

```
user = users.get_current_user()

if user:
  print "Welcome, %s!" % user.nickname()
  if users.is_current_user_admin():
    print "<a href=\"/admin/\">Go to admin area</a>"
```

提示：要将应用程序的某部分限制为只有管理员可以访问，有一种简单的方法就是使用网址处理程序的 login: admin 配置元素。请参阅配置应用程序。

User 类

User 类的实例代表具有 Google 帐户的一个用户。

User 由 google.appengine.api.users 模块提供。

简介

User 类的一个实例代表具有 Google 帐户的一个用户。通过在没有参数的情况下调用构造函数或调用 users.get_current_user() 函数，应用程序可获取代表当前登录到该应用程序的用户的 User 实例。如果该当前用户未登录，User 构造函数将抛出 UserNotFoundError。（如果该用户未登录，users.get_current_user() 将不会抛出异常。）

User 对象是可比较的。如果两个 User 对象相同，那么这两个对象代表同一用户。

User 对象可以是数据库实体属性的值。请参阅类型和 Property 类。

注意：用户可以更改 Google 帐户的电子邮件地址。请参阅 User 对象中的注释。

构造函数

class User(email=None)

代表具有 Google 帐户的用户。

参数：

email

期望的用户的电子邮件地址。如果忽略此项，该对象将代表当前用户（发出请求的用户）。如果未指定任何电子邮件地址，且当前用户未登录，将抛出 UserNotFoundError。

创建 User 对象时，不会检查该电子邮件地址是否有效。如果 User 对象具有的电子邮件地址与有效 Google 帐户不相符，则该 User 对象仍可存储在数据库中，但其永远不会与真正的用户相匹配。

实例方法

User 实例可提供以下方法：

nickname()

返回用户的 [昵称]，这是一个可显示的名称。用户更改其昵称的功能尚未实现，但应用程序目前可使用此功能更改

可显示的名称并能通过实现此功能获益。

对于没有自定义昵称的用户，如果用户的电子邮件地址与应用程序位于同一域内，则其昵称将为该电子邮件地址的 [名称] 部分，否则将为该用户的完整电子邮件地址。

email()

返回用户的电子邮件地址。应用程序应使用昵称作为可显示的名称。

函数

google.appengine.api.users 包提供以下函数：

create_login_url(dest_url)

返回一个网址，当访问该网址时，将提示用户使用 Google 帐户登录，然后将用户重定向回指定为 dest_url 的网址。该网址适用于链接、按钮和重定向。

dest_url 可以是完整的网址，也可以是相对于您的应用程序的域的路径。

create_logout_url(dest_url)

返回一个网址，当访问该网址时会注销用户，然后将用户重定向回指定为 dest_url 的网址。该网址适用于链接、按钮和重定向。

dest_url 可以是完整的网址，或相对于您的应用程序的域的路径。

get_current_user()

如果用户已登录，返回当前用户（提出了正被处理的请求的用户）的 User 对象；如果用户未登录，返回 None。

is_current_user_admin()

如果当前用户已登录且当前注册为该应用程序的管理员，则返回 True。

异常

google.appengine.api.users 包提供以下 exception 类：

exception Error()

这是该包中所有异常的基类。

exception UserNotFoundError()

如果没有提供 email_address，且当前用户未登录，则由 User 构造函数抛出异常。

exception RedirectTooLongError()

给予 create_login_url() 或 create_logout_url() 的重定向网址超过了重定向网址允许的最大长度。

webapp 框架

webapp 框架是一种简单的与 WSGI 兼容的网络应用程序框架，可以与 App Engine 配合使用。不必为了使用 App Engine 而使用 webapp：网络服务器支持任何使用 CGI 的 Python 应用程序。webapp 提供一种简单的方式来开始为 App Engine 开发应用程序。

本参考包括以下小节：

概述

App Engine 支持任何采用 CGI 接口的 Python 应用程序。网络应用程序框架可以通过处理接口的详细信息来简化开发过程，让您更加专注在应用程序功能的设计上。App Engine 包括一种名为 webapp 的简单网络应用程序框架。

webapp 是一种与 WSGI 兼容的框架。您可以使用 CGI 适配器（例如 Python 标准库提供的适配器）将 webapp 或任何其他 WSGI 框架与 App Engine 配合使用。

此处是一个将 CGI 适配器与 App Engine 配合使用的简单 webapp 应用程序：

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

class MainPage(webapp.RequestHandler):
  def get(self):
    self.response.headers['Content-Type'] = 'text/plain'
```

```
    self.response.out.write('Hello, webapp World!')

application= webapp.WSGIApplication([('/', MainPage)],
                                    debug=True)

def main():
  run_wsgi_app(application)

if __name__== "__main__":
  main()
```

运行应用程序

一个 webapp 应用程序由三部分组成：

- 一个或多个 RequestHandler 类（如请求处理程序中所述）
- 将网址映射至 RequestHandler 类的 WSGIApplication 对象
- 使用 CGI 适配器运行 WSGIApplication 的一个主要例程

WSGIApplication 类可以实现 WSGI 接口，该接口是网络应用程序框架和网络服务器之间的标准接口。使用 WSGI CGI 适配器可让任何 WSGI 框架与 App Engine 配合使用。webapp 包含这样一个适配器：函数 run_wsgi_app() 使用应用程序实例并运行此实例。您也可以使用包含在 Python 标准库的 wsgiref 模块中的 CGI 适配器。

以下示例将四个网址路径映射到四个 RequestHandler 类（未显示），然后使用 run_wsgi_app() 运行该应用程序：

```
from google.appengine.extimport webapp
from google.appengine.ext.webapp.utilimport run_wsgi_app

application= webapp.WSGIApplication([('/', MainPage),
                                     ('/newentry', NewEntry),
                                     ('/editentry', EditEntry),
                                     ('/deleteentry', DeleteEntry),
                                     ],
                                     debug=True)

def main():
  run_wsgi_app(application)
```

WSGIApplication 构造函数使用将网址路径映射到 RequestHandler 类的对（元组）列表。

可选的 debug=True 参数可将应用程序设置为调试模式，这样可通知 webapp 在处理程序抛出异常时在浏览器中显示堆栈跟踪。默认情况下，当发生错误时，webapp 只返回一个 HTTP 500 错误。

网址映射

映射中的网址路径是一个正则表达式。必须转义正则表达式特殊字符。正则表达式可以包含正则表达式分组以与网址部分相匹配。将分组中相匹配的模式作为参数传递至请求处理程序。

```
class BrowseHandler(webapp.RequestHandler):

  def get(self, category, product_id):
    # Display product with given ID in the given category.


# Map URLs like /browse/(category)/(product_id) to BrowseHandler.
application= webapp.WSGIApplication([(r'/browse/(.*)/(.*)', BrowseHandler)
                                     ],
                                     debug=True)

def main():
  run_wsgi_app(application)
```

提示：App Engine 基于在应用程序的 app.yaml 文件中所指定的网址和映射将请求传送至 Python 脚本。webapp WSGIApplication 会进一步将具体的网址路径映射至请求处理程序。如何使用这两种映射全由您决定：您可以将所有非静态网址映射至单个 Python 脚本，然后使该脚本将所有动态网址调度到处理程序。您也可以将功能分组成由不同脚本运行的多个 WSGI 应用程序，然后使用 app.yaml 将相应的网址映射至相应的应用程序。

请求处理程序

WSGIApplication 在收到请求时，会创建一个与请求中网址路径相关联的 RequestHandler 类的实例。然后，调用与请求的 HTTP 操作相对应的方法，例如 HTTP GET 请求的 get() 方法。该方法会处理请求并准备响应，然后返回。最后，应用程序将响应发送到客户端。

以下示例定义了响应 HTTP GET 请求的请求处理程序：

```
class AddTwoNumbers(webapp.RequestHandler):

  def get(self):
    try:
      first= int(self.request.get('first'))
      second= int(self.request.get('second'))

      self.response.out.write("<html><body><p>%d + %d = %d</p></body></html>" %
                              (first, second, first+ second))
    except (TypeError, ValueError):
      self.response.out.write("<html><body><p>Invalid inputs</p></body></html>")
```

请求处理程序可定义以下任何方法以处理相应的 HTTP 操作：

- get()
- post()
- head()
- options()
- put()
- delete()
- trace()

请求数据

请求处理程序实例可使用请求数据的 request 属性访问该请求数据。该数据由应用程序初始化为填充的 WebOb Request 对象。

该 Request 对象可提供 get() 方法，此方法会返回从该查询以及 POST 数据中解析的参数值。该方法使用参数名称作为其第一个参数。例如：

```
class MyHandler(webapp.RequestHandler):
  def post(self):
      name= self.request.get("name")
```

默认情况下，如果请求的参数不在该请求中，get() 则会返回空字符串 (")。如果指定了 default_value 参数，get() 会返回该参数的值，而不是返回参数不存在时的空字符串。

如果参数在一个请求中出现多次，默认情况下，get() 会返回其第一次出现时的值。要以列表（可能为空）的形式获取可能出现多次的参数每次出现时的值，请赋予 get() 参数 allow_multiple=True。

```
# <input name="name" type="text" />
name= self.request.get("name")

# <input name="subscribe" type="checkbox" value="yes" />
subscribe_to_newsletter= self.request.get("subscribe", default_value="no")

# <select name="favorite_foods" multiple="true">...</select>
favorite_foods= self.request.get("favorite_foods", allow_multiple=True)
for foodin favorite_foods:
  # ...
```

对于正文内容不是一组 CGI 参数的请求（例如 HTTP PUT 请求的正文），该 Request 对象会提供属性 body 和 body_file。body 是字节字符串形式的正文内容。body_file 提供指向相同数据的类似于文件的接口。

```
uploaded_file= self.request.body
```

WebOb 是开源第三方库。有关 API 参考和示例的详细信息，请参阅 WebOb 文档。

构建响应

请求处理程序实例使用其 response 属性构建响应。这由应用程序初始化为空 Response 对象。

Response 对象的 out 属性是一种类似文件的对象，可以用于编写响应的主体。

```
class MyHandler(webapp.RequestHandler):
  def get(self):
    self.response.out.write("<html><body><p>Hi there!</p></body></html>")
```

[out] 流会缓冲内存中的所有输出，然后在处理程序退出时发送最终输出。webapp 不支持到客户端的流数据。

[clear()] 方法会清除输出缓冲的内容，将其留空。

如果写入到输出流的数据为 Unicode 值，或者响应包含以 ; charset=utf-8 结尾的 [Content-Type] 标头，则 webapp 会将输出编码为 UTF-8。默认情况下，[Content-Type] 标头为 text/html; charset=utf-8 格式，包括编码行为。如果 [Content-Type] 更改为采用不同的 charset，则 webapp 会假设输出为要逐字发送的字节字符串。

注意：虽然 Request 类是从 WebOb 库衍生的，但 Response 类不是。不过，Response 类包含多种行为类似等效 WebOb 的方法。（以后的 API 版本可能直接使用 WebOb 类。）

重定向、标头以及状态代码

通常，响应会包含 HTTP 状态代码 200，表示 [正常]。具体而言，代码 200 表明 URI 引用一个有效资源，且该资源包含于响应的输出中。不同的环境调用不同的错误代码。例如，如果存在一个服务器的内部错误条件，且该错误条件阻止了预期数据的 输出，那么，该服务器可能会返回一个代码 500，表示 [服务器错误]。

该请求处理程序会提供一种 error(...) 方法以使用指定的错误代码准备错误响应。例如：

```
class MyHandler(webapp.RequestHandler):
  def get(self):
    self.response.out.write("You asked me to do something.")
    try:
        doSomething()
      self.response.out.write("It's done!")

    except Error:
      # Clear output and return an error code.
      self.error(500)
```

error(...) 方法会使用数字 HTTP 状态代码，并准备该请求处理程序的响应以使用此状态代码。它还会清除输出缓冲区，这样，处理程序便可以通过使用 out 准备顺利输出，之后如果有问题再调用 error(...)。

状态代码的另一个常见用途是将用户的浏览器重定向至另一个 URI。重定向可为永久性的，表示该 URI 曾经对所请求的资源有效，但该资源以后的所有请求应当使用新的 URI。重定向也可为临时的，表示所请求的 URI 是有效的，但浏览器目前应当请求另一个 URI。网络应用程序的常见技术是使用临时的重定向对表单成功提交做出响应，从而防止用户意外使用浏览器的 [返回] 按钮然后重新提交该表单。

请求处理程序提供了一个 redirect(...) 方法以准备重定向响应。例如：

```
class FormHandler(webapp.RequestHandler):
  def post(self):
    if processFormData(self.request):
      self.redirect("/home")
    else:
      # Display the form, possibly with error messages.
```

redirect(...) 使用目标 URI 作为其第一个参数。默认情况下，它会建立一个临时重定向。可选参数 permanent=True 使用永久重定向代码。

请求处理程序方法 error(...) 和 redirect(...) 会更改响应的 HTTP 状态代码。redirect(...) 也可使用 HTTP 标头将新的 URI 传递到客户端。Response 对象可提供用来直接设置状态代码和 HTTP 标头的方法。

Response 对象的 set_status(...) 方法可更改响应的状态代码。该方法使用数字状态代码作为其第一个参数。可选的第二个参数指定了将代替指定状态代码默认邮件的邮件。

Response 对象的 headers 属性是一个 wsgiref.headers.Headers 实例，代表该响应的 HTTP 标头。有关如何设置标头的信息，请参阅 wsgiref.headers 文档。

```
class StatusImageHandler(webapp.RequestHandler):
  def get(self):
     img_data= get_status_image_for_current_user()
    self.response.headers["Content-Type"] = "image/png"
    self.response.headers.add_header("Expires", "Thu, 01 Dec 1994 16:00:00 GMT")
    self.response.out.write(img_data)
```

Request 类

Request 类的实例包含有关传入的网络请求的信息。

Request 由 google.appengine.ext.webapp 模块提供。

Request 类继承自 WebOb Request 类。这里只讨论 WebOb Request 类的几个功能。有关详细信息，请参阅 WebOb 文档。

- 继承自 WebOb Request 的实例变量：
  - body
  - body_file
  - remote_addr
  - url
  - path
  - query_string
  - headers
  - cookies

简介

由 webapp 提供的 Request 类继承自 WebOb Request 类。webapp 添加了多种新方法以访问由网络表单提交的参数，且可扩展多项默认行为。

```
class MyRequestHandler(webapp.RequestHandler):
  def get(self):
    self.response.out.write('''
     <html>
       <body>
         <form action="post">
           <p>Name: <input type="text" name="name" /></p>
           <p>Favorite foods:</p>
           <select multiple size="4">
             <option value="apples">Apples</option>
             <option value="bananas">Bananas</option>
             <option value="carrots">Carrots</option>
             <option value="durians">Durians</option>
           </select>
           <p>Birth year: <input type="text" name="birth_year" /></p>
         </form>
       </body>
     </html>
     ''')

  def post(self):
      name= self.request.get("name")
      favorite_foods= self.request.get_all("favorite_foods")
      birth_year= self.request.get_range("birth_year",
                                         min_value=1900,
                                         max_value=datetime.datetime.utcnow().year,
                                         default_value=1900)
```

除了下面所描述的几种新方法外，webapp Request 类与 WebOb Request 还有以下区别：

- 如果未在该请求的内容类型标头中指定字符集，则假定 UTF-8 为其字符集。
- 忽略将表单参数解码为 Unicode 时发生的错误。构造函数会强制为 WebOb Request 构造函数分配以下参数：unicode_errors='ignore'
- 表单参数键会使用与值相同的字符集予以解码。构造函数会强制为 WebOb Request 构造函数分配以下参数: decode_param_names=True

构造函数

Request 类的构造函数的定义如下：

class Request(environ)

对 webapp 应用程序的传入的请求。通常，WSGIApplication 会实例化一个 RequestHandler 并通过一个用 WSGI 兼容的环境参照表 (environ) 填充的 Request 对象对其进行初始化。

参数：

environ
WSGI 兼容的环境参照表。

实例方法

Request 类提供以下实例方法：

get(argument_name, default_value='')

返回查询（网址）的值或具有指定名称的 POST 参数。如果多个参数具有同一个名称，会返回第一个参数的值。网址和请求正文会采用网络浏览器用来提交表单的标准格式。

参数：

argument_name
要获取的参数名称。
default_value
指定名称的参数不存在时该方法应当返回的值。默认为空字符串。

get_all(argument_name)

返回所有查询（网址）的值的列表或具有指定名称的 POST 参数，可能返回一个空列表。

参数：

argument_name
要获取的参数名称。

arguments()

返回查询（网址）的名称列表或 POST 数据参数。即使数据包含具有同一名称的多个参数，该参数名称也仅在列表中显示一次。

get_range(name, min_value=None, max_value=None, default=0)

将该查询（网址）或具有指定名称的 POST 数据参数解析为 int 并将其返回。该值会进行标准化，以适合指定范围（如果有的话）。

参数：

name
: 要作为整数获取的参数名称。

min_value
: 该参数的最小值。如果值小于此最小值，该方法会返回最小值。

max_value
: 该参数的最大值。如果值大于此最大值，该方法会返回最大值。

default
: 指定名称的参数不存在时要返回的值。

继承自 WebOb Request 的实例变量

以下是继承自 WebOb Request 类的实例变量成员的部分列表。有关详细信息，请参阅 WebOb 文档。

body
: 请求正文，字节字符串形式。

body_file
: 请求正文，StringIO 实例形式（类似于文件的对象）。

remote_addr
: 远程用户的 IP 地址。

url
: 完整请求网址。

path
: 在托管名称和查询参数之间的网址路径。

query_string
: 网址的查询参数，第一个 ？ 后的全部内容。

headers
: 请求标头，是一种类似于参照表的对象。Key 不区分大小写。

cookies
: 请求中的 cookie 数据，是一种类似于参照表的对象。

Response 类

Response 类的实例表示在对网络请求的响应中要发送的数据。

Response 由 google.appengine.ext.webapp 模块提供。

简介

webapp 框架调用请求处理程序方法时，会使用一个空 Response 实例初始化该处理程序实例的 response 成员。该处理程序方法通过处理 Response 实例（例如将正文数据写入 out 成员或在 headers 成员上设置标头）准备响应。

```
import datetime

class MyRequestHandler(webapp.RequestHandler):
  def get(self):
    self.response.out.write("<html><body>")
    self.response.out.write("<p>Welcome to the Internet!</p>")
    self.response.out.write("</body></html>")

      expires_date= datetime.datetime.utcnow() + datetime.timedelta(365)
      expires_str= expires_date.strftime("%d %b %Y %H:%M:%S GMT")
    self.response.headers.add_header("Expires", expires_str)
```

当处理程序方法返回时，webapp 会发送该响应。当该方法返回时，该响应的内容为 Response 对象的最终状态。

注意：用处理程序方法处理对象时不会将任何数据传递给用户。特别要说明的是，这表示 webapp 无法像在流应用程序中那样向浏览器发送数据然后执行其他逻辑操作。（无论是否使用 webapp，App Engine 应用程序都无法向浏览器传送数据。）

默认情况下，响应会使用 HTTP 状态代码 200（[正常]）。应用程序可使用 set_status() 方法更改该状态代码。有关设置错误代码的便捷方法，您也可参阅 RequestHandler 对象的 error() 方法。

如果响应未在 Content-Type 标头中指定字符集，则该响应的字符集会自动设置为 UTF-8。

构造函数

Response 类的构造函数的定义如下：

class Response()

输出的响应。通常，WSGIApplication 会实例化 RequestHandler，并使用带有默认值的 Response 对象将其初始化。

类方法

Response 类会提供以下类方法：

Response.http_status_message(code)

返回指定 HTTP 状态代码的默认 HTTP 状态消息。

参数：

code
HTTP 状态代码。

实例变量

Response 类的实例具有以下变量成员：

out

包含响应的正文文本的 StringIO 类的实例。返回请求处理程序方法时，该对象的内容会作为响应的正文发送。

headers

包含响应标头的 wsgiref.headers.Headers 类的实例。返回请求处理程序方法时，该对象的内容会作为响应的标头发送。

出于安全性考虑，应用程序不能修改某些响应标头。请参阅禁止使用的 HTTP 响应标头。

实例方法

Response 类的实例具有以下方法：

set_status(code, message=None)

为响应设置 HTTP 状态代码。

参数：

code
: 要用于响应的 HTTP 状态代码。

message
: 要与 HTTP 状态代码配合使用的消息。如果是 None，则使用如 Response.http_status_message() 返回的默认消息。

clear()

清除写入输出流 (out) 的所有数据。

wsgi_write(start_response)

使用 WSGI 语义编写响应。通常，应用程序不直接调用此函数。当请求处理程序方法返回时，webapp 会调用此函数以编写响应。

参数：

start_response
: WSGI 兼容的 start_response 函数。

禁止使用的 HTTP 响应标头

出于安全性考虑，应用程序不能修改以下 HTTP 响应标头。在 Response 对象的 headers 对象中设置这些响应标头会无效。

- Content-Encoding
- Content-Length
- Date
- Server
- Transfer-Encoding

RequestHandler 类

RequestHandler 类是 HTTP 请求处理程序的超类。

RequestHandler 由 google.appengine.ext.webapp 模块提供。

简介

webapp 应用程序会定义一个或多个 RequestHandler 类以处理请求。handler 类会覆盖以下一个或多个方法以处理相应类型的 HTTP 请求：get()、post()、head()、options()、put()、delete() 或 trace()。

构造函数

class RequestHandler()

HTTP 请求处理程序的基类。

该构造函数不使用任何参数。可使用 initialize() 方法初始化该实例。

实例方法

RequestHandler 类的子类可继承或覆盖以下方法：

get(*args)
    调用此方法以处理 HTTP GET 请求。可由 Handler 子类覆盖此方法。
post(*args)
    调用此方法以处理 HTTP POST 请求。可由 Handler 子类覆盖此方法。
put(*args)
    调用此方法以处理 HTTP PUT 请求。可由 Handler 子类覆盖此方法。
head(*args)
    调用此方法以处理 HTTP HEAD 请求。可由 Handler 子类覆盖此方法。
options(*args)
    调用此方法以处理 HTTP OPTIONS 请求。可由 Handler 子类覆盖此方法。
delete(*args)
    调用此方法以处理 HTTP DELETE 请求。可由 Handler 子类覆盖此方法。
trace(*args)
    调用此方法以处理 HTTP TRACE 请求。可由 Handler 子类覆盖此方法。
handle_exception(exception, debug_mode)
    处理程序抛出异常时调用此方法。默认情况下，handle_exception 会将 HTTP 状态代码设置为 500（[服务器错

误])。如果 debug_mode 为 True，则它将向浏览器打印一个堆栈跟踪。否则，将只打印一个纯文本错误消息。RequestHandler 类可覆盖此方法以提供自定义行为。

error(code)

处理程序用来返回错误响应的一种快捷方法。清除响应输出流并将 HTTP 错误代码设置为 code。这相当于调用 self.response.clear() 和 self.response.set_status(code)。

redirect(uri, permanent=False)

处理程序用来返回重定向响应的一种快捷方法。设置 HTTP 错误代码和 Location: 标头以重定向至 uri，并清除响应输出流。如果 permanent 为 True，则它将使用 HTTP 状态代码 301 来进行永久重定向。否则，将使用 HTTP 状态代码 302 进行临时重定向。

initialize(request, response)

使用 Request 和 Response 对象初始化处理程序实例。通常，WSGIApplication 在实例化 Handler 类后执行此操作。

实例属性

RequestHandler 的子类的实例具有以下属性：

request

Request 实例。通常，由 WSGIApplication 在构建对象之后初始化该属性。

response

Response 实例。通常，由 WSGIApplication 在构建对象之后初始化该属性。

WSGIApplication 类

google.appengine.ext.webapp 包提供以下类：

class WSGIApplication(url_mapping, debug=False)

将网址路径映射到 RequestHandler 类的与 WSGI 兼容的应用程序。App Engine CGI 脚本可以创建 WSGIApplication 对象，然后使用 WSGI CGI 处理程序（例如 run_wsgi_app()）运行此对象。

参数：

url_mapping

从网址路径到请求处理程序的映射，形式为元组列表，其中每个元组都是从路径到处理程序的映射。该元组的第一元素以字符串的形式指定网址路径。该元组的第二元素是请求 Handler 类（RequestHandler 的子类）的构造函数。

debug

如果为 True，该应用程序将在 [调试模式] 下运行。从根本上讲，这表示当处理程序抛出异常时，该请求处理程序的 handle_exception() 方法将与 debug_mode=True 一起调用，这样可以将调试信息打印到网络浏览器。

实用工具函数

google.appengine.webapp.util 包提供以下函数：

@login_required

webapp 请求处理程序方法的 Python 注释，用于验证用户是否使用 Google 帐户登录，如果未登录，会将该用户重定向至登录页面。登录页面将重定向至请求网址。

此注释仅适用于 GET 请求。

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import login_required

class MyHandler(webapp.RequestHandler):

  @login_required
  def get(self):
      user = users.GetCurrentUser(self)
     self.response.out.write('Hello, ' + user.nickname())


run_wsgi_app(application)
```

在 App Engine 的 CGI 环境中运行 WSGI 应用程序。这与使用从 WSGI 到 CGI 的适配器（例如 Python 标准库的 wsgiref 模块提供的适配器）类似，然而相比之下，前者具有以下几个优势：run_wsgi_app() 可以自动检测应用程序是否在开发服务器中运行，如果在其中运行，则会将错误写入日志，而且还会输出到浏览器中。run_wsgi_app() 还允许 App Engine 在发送错误前发送处理程序输出的数据，而 wsgiref.handlers.CGIHandler 并不执行此操作。

参数：

application
WSGI 应用程序对象（例如 webapp.WSGIApplication）。

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

class MainPage(webapp.RequestHandler):
  def get(self):
     self.response.headers['Content-Type'] = 'text/plain'
     self.response.out.write('Hello, webapp World!')

application = webapp.WSGIApplication([('/', MainPage)],
                                     debug=True)

def main():
   run_wsgi_app(application)
```

```
if __name__== "__main__":
    main()
```

配置应用程序

App Engine 应用程序在其根目录中有名为 app.yaml 的配置文件。该文件介绍了应用程序所使用的 App Engine 运行时环境、源代码的版本，以及哪些处理程序脚本应处理哪些网址。app.yaml 和应用程序源代码一起上传到 App Engine。

- 必需的元素
- 脚本处理程序
- 静态文件的处理程序
- 需要登录或管理员身份
- 跳过文件
- 参考 Python 库目录
- 保留的网址

以下是 app.yaml 文件的示例：

```
application: myapp

version: 1

runtime: python

api_version: 1



handlers:

- url: /

    script: home.py



- url: /index\.html

    script: home.py



- url: /stylesheets
```

```
  static_dir: stylesheets


- url: /(.*\.(gif|png|jpg))
  static_files: static/\1
  upload: static/(.*\.(gif|png|jpg))


- url: /admin/.*
  script: admin.py
  login: admin


- url: /.*
  script: not_found.py
```

app.yaml 的语法为 YAML 格式。有关该语法的详细信息，请参阅 YAML 网站。

提示：YAML 格式支持注释。以井号 (#) 字符开头的行会被忽略：
This is a comment.

网址和文件路径样式使用 POSIX 扩展的正则表达式语法，对照元素和对照类除外。支持分组匹配项的反向引用（例如 \1），正如支持以下 Perl 扩展一样：\w \W \s \S \d \D（这类似于 Codesite 搜索，增加了反向引用支持）。

必需的元素

app.yaml 文件必须包括以下各元素中的一个：

application

应用程序标识符。这是当您在管理控制台中创建应用程序时选定的标识符。

application: myapp

version

应用程序代码的版本分类符。App Engine 为使用的每个 version 保留一份应用程序副本。管理员可以使用管理控制台更改应用程序的哪个主版本是公共的，并可以先测试非公共版本，再将非公共版本变为公共版本。

应用程序的每个版本都会保留其自身的 app.yaml 副本。当上传应用程序时，将通过上传创建或替换正在上传的 app.yaml 文件中提到的版本。

在管理控制台上，版本号显示为主版本 (version) 以及与此主版本上传次数相对应的次版本。只有最新的主版本才会保留下来。

version: 1

runtime

该应用程序使用的 App Engine 运行时环境的名称。此时，App Engine 有一个运行时环境：python

runtime: python

api_version

该应用程序在指定运行时环境中使用的 API 的版本。当 Google 发布运行时环境 API 的新版本时，您的应用程序会继续使用针对该应用程序编写的 API。要将您的应用程序升级到新 API，请更改该值并上传已升级的代码。

此时，App Engine 有一个 python 运行时环境的版本：1。

api_version: 1

handlers

网址样式及其处理方式说明的列表。App Engine 可以通过执行应用程序代码，或通过提供与代码一起上传的静态文件（例如图像、CSS 或 JavaScript）来处理网址。

根据样式在 app.yaml 中的显示顺序从上到下对其进行评估。样式与网址匹配的第一个映射将用于处理请求。

有两种处理程序：脚本处理程序和静态文件处理程序。脚本处理程序在应用程序中运行 Python 脚本以确定指定网址的响应。静态文件处理程序返回文件的内容（例如图像）作为响应。

有关该值的详细信息，请参阅下面的脚本处理程序以及静态文件的处理程序。

handlers:

- url: /images

  static_dir: static/images

- url: /.*

  script: myapp.py

脚本处理程序

脚本处理程序执行 Python 脚本以处理与网址样式匹配的请求。映射定义要匹配的网址样式和要执行的脚本。

url

网址样式，作为正则表达式。表达式可以通过正则表达式反向引用包含可在脚本的文件路径中参考的分组。

例如，/profile/(.*?)/(.*) 可能与网址 /profile/edit/manager 匹配，并使用 edit 和 manager 作为第一个和第二个分组。

handlers:

- url: /profile/(.*?)/(.*)

  script: /employee/\2/\1.py

script

脚本的路径，从应用程序根目录开始。

如以上示例所示，对于 edit 和 manager 分组，脚本路径 /employee/\2/\1.py 使用完整路径 /employee/manager/edit.py。

以下示例将网址映射到脚本：

handlers:

The root URL (/) is handled by the index.py script. No other URLs match this pattern.

- url: /

  script: index.py

The URL /index.html is also handled by the index.py script.

```
- url: /index\.html
  script: index.py

# A regular expression can map parts of the URL to the file path of the script.
- url: /browse/(books|videos|tools)
  script: \1/catalog.py

# All other URLs use the not_found.py script.
- url: /.*
  script: not_found.py
```

静态文件的处理程序

静态文件是直接向指定网址的用户提供的文件，例如图像、CSS 样式表或 JavaScript 源文件。静态文件处理程序说明了应用程序目录中的哪些文件为静态文件，以及为哪些网址提供静态文件。

为了提高效率，App Engine 将应用程序文件与静态文件单独存储和提供。静态文件在应用程序的文件系统中不可用。如果您有需要由应用程序代码读取的数据文件，数据文件必须为应用程序文件，且必须和静态文件样式不匹配。

除非另有通知，否则网络浏览器会将从网站上加载的文件保留有限的一段时间。您可以通过包括 default_expiration 元素（顶级元素），来定义应用程序的所有静态文件处理程序的全局默认缓存周期。您还可以为特定静态文件处理程序配置缓存持续时间。（脚本处理程序可以通过向浏览器返回相应的 HTTP 标头来设置缓存持续时间。）

default_expiration

如果处理程序没有指定自己的 expiration，则静态文件处理程序提供的静态文件的时间长度应缓存在用户的浏览器中。值是一串数字和单位，由空格分隔，其中单位可以用 d 代表天、h 代表小时、m 代表分钟、s 代表秒。例如，"4d 5h" 将缓存期设置为从浏览器首次加载文件开始算起的 4 天 5 小时。

default_expiration 为可选项。如果被忽略，默认行为是允许浏览器确定其自身的缓存持续时间。

例如：

```
application: myapp
```

```
version: 1

runtime: python

api_version: 1



default_expiration: "4d 5h"



handlers:

  # ...
```

可以用两种方式定义静态文件处理程序：作为映射到网址路径的静态文件目录结构，或作为将网址映射到特定文件的样式。

静态目录处理程序

使用静态目录处理程序可以轻松将目录的全部内容作为静态文件来提供。除非被目录的 mime_type 设置覆盖，否则每个文件都使用与其文件扩展名对应的 MIME 类型提供。指定目录中的所有文件会作为静态文件上传，其中没有文件可以作为脚本运行。

url

网址前缀。该值使用正则表达式语法（因此必须对 regexp 特殊字符进行转义），但它不应该包含分组。所有以该前缀开头的网址都由该处理程序处理，将前缀后面的部分网址用作文件路径的一部分。

static_dir

包括静态文件的目录的路径，从应用程序根目录开始。匹配的 url 样式末尾的所有内容附加到 static_dir 以形成到请求的文件的完整路径。

该目录中的所有文件都作为静态文件由应用程序上传。

mime_type

可选。如果指定，将使用指定的 MIME 类型提供该处理程序提供的所有文件。如果未指定，文件的 MIME 类型将取自该文件的文件扩展名。

有关可以使用的 MIME 媒体类型的详细信息，请参阅 IANA MIME 媒体类型网站。

expiration

该处理程序提供静态文件的时间长度应在用户的浏览器中进行缓存。值是一串数字和单位，由空格分隔，其中单位可以用 d 代表天、h 代表小时、m 代表分钟、s 代表秒。例如，"4d 5h" 将缓存期设置为从浏览器首次加载文件开始算起的 4 天 5 小时。

expiration 为可选项。如果被忽略，将使用应用程序的 default_expiration。

例如：

handlers:

```
# All URLs beginning with /stylesheets are treated as paths to static files in

# the stylesheets/ directory.  Note that static_dir handlers do not use a

# regular expression for the URL pattern, only a prefix.

- url: /stylesheets

  static_dir: stylesheets
```

静态文件样式处理程序

静态文件处理程序将网址样式与使用应用程序上传的静态文件的路径相关联。网址样式正则表达式可以定义在文件路径的结构中使用的正则表达式分组。您可以使用它而非 static_dir 来映射到目录结构中的特定文件，而不是映射整个目录。

静态文件不能与应用程序代码文件相同。如果静态文件路径与动态处理程序中使用的脚本的路径匹配，则该动态处理程序将无法使用此脚本。

以下 static_dir 和 static_files 处理程序对等：

```
- url: /images

  static_dir: static/images


- url: /images/(.*)

  static_files: static/images/\1
```

```
  upload: static/images/(.*)
```

url

网址样式，作为正则表达式。表达式可以通过正则表达式反向引用包含可在脚本的文件路径中参考的分组。

例如，/item-(.*?)/category-(.*) 可能与网址 /item-127/category-fruit 匹配，并使用 127 和 fruit 作为第一个和第二个分组。

```
handlers:

- url: /item-(.*?)/category-(.*)

  static_files: archives/\2/items/\1
```

static_files

与网址样式匹配的静态文件的路径，从应用程序根目录开始。路径可以参考网址样式的分组中匹配的文本。

如以上示例所示，archives/\2/items/\1 分别在 \2 和 \1 位置插入匹配的第二个和第一个分组。采用以上示例中的样式，文件路径应当为 archives/fruit/items/127。

upload

与该处理程序将引用的所有文件的文件路径匹配的正则表达式。该表达式是必需的，因为处理程序无法确定您的应用程序目录中哪些文件与指定 url 和 static_files 样式相对应。静态文件的上传和处理与应用程序文件相独立。

以上示例可能使用以下 upload 样式：archives/(.*?)/items/(.*)

mime_type

可选。如果指定，将使用指定的 MIME 类型提供该处理程序提供的所有文件。如果未指定，文件的 MIME 类型将取自该文件的文件扩展名。

有关可以使用的 MIME 媒体类型的详细信息，请参阅 IANA MIME 媒体类型网站。

expiration

该处理程序提供静态文件的时间长度应在用户的浏览器中进行缓存。值是一串数字和单位，由空格分隔，其中单位可以用 d 代表天、h 代表小时、m 代表分钟、s 代表秒。例如，"4d 5h" 将缓存期设置为从浏览器首次加载文件开始算起的 4 天 5 小时。

expiration 为可选项。如果被忽略，将使用应用程序的 default_expiration。

例如：

handlers:

```
# All URLs ending in .gif .png or .jpg are treated as paths to static files in
# the static/ directory.  The URL pattern is a regexp, with a grouping that is
# inserted into the path to the file.
- url: /(.*\.(gif|png|jpg))
  static_files: static/\1
  upload: static/(.*\.(gif|png|jpg))
```

需要登录或管理员身份

任何网址处理程序都可以使用 login 设置将访问者限制为仅登录了的用户，或是应用程序管理员的用户。当具有 login 设置的网址处理程序与网址匹配时，该处理程序会先检查用户是否用 Google 帐户登录了应用程序。如果没有，用户将被重定向到 Google 登录页面，并在登录或创建帐户后重定向回该应用程序网址。

如果设置为 login: required，用户登录后，处理程序将正常运行。

如果设置为 login: admin，用户登录后，处理程序将检查用户是否是应用程序的管理员。如果不是，将向用户显示错误消息。如果用户是管理员，处理程序将继续运行。

如果应用程序需要其他行为，应用程序可以自行执行用户处理。有关详细信息，请参阅用户 API。

示例：

handlers:

```
- url: /profile/.*
  script: user_profile.py
  login: required
```

```
- url: /admin/.*
  script: admin.py
  login: admin

- url: /.*
  script: welcome.py
```

跳过文件

应用程序目录中路径与 static_dir 路径或 static_files upload 路径匹配的文件被视为静态文件。应用程序目录中的所有其他文件均被视为应用程序和数据文件。

skip_files 元素指定应用程序目录中的哪些文件不上传到 App Engine。值为正则表达式。当上传应用程序时，将从要上传的文件列表中忽略与正则表达式匹配的任何文件名。

skip_files 具有以下默认值：

```
skip_files: |
 ^(.*/)?(
 (app\.yaml)|
 (app\.yml)|
 (index\.yaml)|
 (index\.yml)|
 (#.*#)|
 (.*~)|
 (.*\.py[co])|
 (.*/RCS/.*)|
```

(\..*)|

)$

默认样式不包括配置文件 app.yaml、app.yml、index.yaml、index.yml（配置单独发送至服务器），具有 #...# 和 ...~、.pyc 和 .pyo 命名格式的 Emacs 备份文件，RCS 版本控制目录中的文件，以及名称以点 (.) 开头的 Unix 隐藏文件。

如果在 app.yaml 中指定新值，它将覆盖该值。要扩展该样式，请将它和扩展名一起复制并粘贴到您的配置。

参考 Python 库目录

您可以使用 $PYTHON_LIB 参考 app.yaml 脚本路径中的 Python 库目录。仅当设置脚本包含在 App Engine 库中的处理程序时，这才有用。

例如，$PYTHON_LIB/google/appengine/ext/admin 是与开发网络服务器的开发人员控制台功能相似的管理应用程序，开发网络服务器自身可以在 App Engine 上作为应用程序的一部分运行。要对其进行设置，请加入使用其路径的脚本处理程序的配置：

handlers:

- url: /admin/.*

  script: $PYTHON_LIB/google/appengine/ext/admin

  login: admin

保留的网址

出于提供功能或管理目的，App Engine 保留了一些网址路径。脚本处理程序和静态文件处理程序路径绝不会与这些路径匹配。

保留了以下网址路径：

- /_ah/
- /form

配置索引

应用程序所进行的每个数据库查询都需要对应的索引。简单查询（例如单个属性的查询）的索引会自动创建。复杂查询的索引必须在名为 index.yaml 的配置文件中进行定义。该文件使用应用程序上传，以便在数据库中创建索引。

当应用程序尝试执行需要索引的查询，但该索引在配置文件中没有相应的条目时，开发网络服务器 (dev_appserver.py) 会自动向该文件添加项目。您可以通过编辑文件调整索引或手动创建新索引。

提示：如果应用程序进行的每个查询都使用开发网络服务器进行了测试，那么在 index.yaml 中生成的条目将是完整的。如果应用程序进行需要索引的查询，而所需的索引未在开发网络服务器上测试，您只需手动编辑文件即可。

有关索引的详细信息，请参阅查询和索引。

以下是 index.yaml 文件的示例：

```
indexes:


- kind: Cat
  ancestor: no
  properties:
  - name: name
  - name: age
    direction: desc


- kind: Cat
  properties:
  - name: name
    direction: asc
  - name: whiskers
    direction: desc
```

```
- kind: Store
  ancestor: yes
  properties:
  - name: business
    direction: asc
  - name: owner
    direction: asc
```

index.yaml 的语法为 YAML 格式。有关该语法的详细信息，请参阅 YAML 网站。

提示：YAML 格式支持注释。以井号 (#) 字符开头的行会被忽略：
This is a comment.

index.yaml 文件格式

index.yaml 具有名为 indexes 的单独列表元素。列表中的每个元素代表应用程序的一个索引。

索引元素可以有以下元素：

kind

查询的实体类型。通常情况下，这是定义实体的模型的 Model 类的名称。该元素为必需的元素。

properties

要作为索引的列包含的属性列表排列顺序如下：先是等效过滤器中使用的属性，后面是不等效过滤器中使用的属性，然后是排列顺序及其方向。

该列表中的每个元素都有以下元素：

name

属性的数据库名称。

direction

排序方向，asc 代表升序，desc 代表降序。只有查询的排列顺序中使用的属性才需要它，且必须与查询使用的方向

匹配。默认为 asc。

ancestor

如果查询有祖先子句（Query.ancestor() 或 GQL ANCESTOR IS 子句），则为 yes。默认为 no。

自动和手动建立索引

当开发网络服务器向 index.yaml 添加生成的索引定义时，它也在以下行执行同样的操作，并在必要时将其插入。

AUTOGENERATED

开发网络服务器将该行下方的所有索引定义视为自动，且可能在应用程序进行查询时更新该行下的现有定义。

该行上方的所有索引定义被视为手动控制，不由开发网络服务器进行更新。网络服务器只会在行下方进行更改，而且只有当整个 index.yaml 文件没有说明应用程序所执行的查询的索引时才这样做。要控制自动索引定义，请将其移到该行上方。

开发网络服务器

App Engine SDK 包括一个用于开发和测试 App Engine 应用程序的网络服务器应用程序。该网络服务器可重现 App Engine Python 运行时环境（包括沙盒限制），并模拟 App Engine 服务（例如，数据库）。

SDK 和网络服务器可在任何装有 Python 2.5 的计算机上运行。您可以在 Python 网站上获取适用于您操作系统的 Python。

如果您尚未拥有 SDK，可以下载 App Engine SDK。

- 运行开发网络服务器
- 使用数据库
- 使用 User
- 使用邮件
- 使用网址抓取
- 开发控制台
- 命令行参数

运行开发网络服务器

当您获得应用程序的目录以及 app.yaml 配置文件后，即可通过 dev_appserver.py 命令启动开发网络服务器：

dev_appserver.py myapp

默认情况下，网络服务器在端口 8080 上进行监听。您可以在以下网址访问该应用程序：http://localhost:8080/

要更改网络服务器使用的端口，请使用 --port 选项：

```
dev_appserver.py --port=9999 myapp
```

要中断网络服务器：在 Windows 环境下，请在命令提示窗口中按 Control-Break 键。在 Mac OS X 或 Unix 环境下，请按 Control-C 键。

该网络服务器在运行时会监视您对文件所做的更改，如有需要，还会重新加载这些文件。对于大多数类型的更改，您只需编辑文件然后在浏览器中重新加载该网页。在某些情况下（例如，当应用程序进行动态导入时），您可能需要重新启动该网络服务器以重设模块导入缓存。

使用数据库

开发网络服务器可以使用您计算机上的一个文件模拟 App Engine 数据库。在网络服务器的调用之间，此文件仍然存在，因此当下一次运行该网络服务器时，您所存储的数据仍将可用。

要清除应用程序的本地数据库，请在启动该网络服务器时使用 --clear_datastore 选项：

```
dev_appserver.py --clear_datastore myapp
```

该网络服务器会在启动时将其正在使用的数据库文件的位置打印到终端。您可以创建该文件的副本，稍后对其进行还原以将该数据库重设为已知状态。请确保在替换数据库文件后重新启动网络服务器。

要更改数据库文件使用的位置，请使用 --datastore_path 选项：

```
dev_appserver.py --datastore_path=/tmp/myapp_datastore myapp
```

当您的应用程序在数据库上执行查询时，开发网络服务器会检查应用程序的 index.yaml 文件是否支持该查询。如果该查询要求在文件中提及其索引，那么服务器将生成一个索引并将该索引添加到此文件中。如果您的应用程序可以尝试执行未进行测试的查询，那么您可能想要编辑此文件。

自创建或上次清除数据库文件以来所执行的每个查询都会生成 index.yaml。查询历史记录会存储在单独的文件中。要更改历史记录文件的位置，请以类似于使用 --datastore_path 选项的方式使用 --history_path 选项。

有关索引和 index.yaml 的详细信息，请参阅查询和索引以及配置索引。

使用 User

开发网络服务器可以使用自己的登录和退出页面模拟 Google 帐户。在开发网络服务器下运行时，users.create_login_url 和 users.create_logout_url 函数将返回本地服务器上的 /_ah/login 和 /_ah/logout 的网址。

开发登录页面包含一个表单，您可以在该表单中输入一个电子邮件地址。不管您输入了什么电子邮件地址，您的会话都会将其作为有效用户使用。

要使应用程序认定登录的用户为管理员，请选中 复选框（位于表单上）。

使用邮件

开发网络服务器可发送电子邮件以调用 App Engine 邮件服务。要启用电子邮件支持，必须为网络服务器提供用于指定要使用的邮件服务器的选项。网络服务器可以使用 SMTP 服务器，也可以使用本地安装的 Sendmail。

要启用使用 SMTP 服务器的邮件支持，请使用具有相应值的 --smtp_host、--smtp_port、--smtp_user 和 --smtp_password 选项。

```
dev_appserver.py --smtp_host=smtp.example.com --smtp_port=25 \
    --smtp_user=ajohnson --smtp_password=k1tt3ns myapp
```

要启用使用 Sendmail 的邮件支持，请使用 --enable_sendmail 选项。网络服务器将使用 sendmail 命令，根据您安装时的默认配置发送电子邮件。

```
dev_appserver.py --enable_sendmail myapp
```

如果未启用使用 SMTP 或 Sendmail 的邮件，那么，尝试从应用程序发送电子邮件的操作将无效，但是在应用程序中则会显示为发送成功。

使用网址抓取

当您的应用程序使用网址抓取 API 创建 HTTP 请求时，开发网络服务器将直接从您的计算机创建该请求。如果您使用代理服务器访问网站，那么，这种行为可能与在 App Engine 上运行应用程序时不同。

注意：dev_appserver.py 一次只能处理一个请求。如果您的应用程序在处理请求的同时对自身创建了网址抓取请求，那么在使用开发网络服务器时，这些请求将会失败。（如果是在 App Engine 上运行，那么请求将不会失败。）要测试这些请求，您可以在其他端口上运行 dev_appserver.py 的另一实例，然后对您的应用程序进行编码以在对其自身创建请求时使用其他服务器。

开发控制台

开发网络服务器包括一个控制台网络应用程序。通过该控制台，您可以浏览本地数据库，并可通过将 Python 代码提交到网络表单与该应用程序进行交互。

要访问控制台，请访问您服务器上的网址 /_ah/admin：http://localhost:8080/_ah/admin

命令行参数

dev_appserver.py 命令支持以下命令行参数：

--datastore_path=...

用于本地数据库数据文件的路径。如果文件不存在，服务器将创建该文件。

--history_path=...

用于本地数据库历史记录文件的路径。

--debug

运行时将详细的调试消息打印到该控制台。

--help

打印有用的消息，然后退出。

--login_url=...

用于用户登录页面的相关网址。默认为 /_ah/login。

--port=...

用于该服务器的端口号。默认为 8080。

--address=...

用于该服务器的主机地址。您可能需要设置该地址，以便能够从网络上的其他计算机访问该开发服务器。地址 0.0.0.0 允许进行本地主机访问和主机名访问。默认为 localhost。

--clear_datastore

启动该网络服务器之前清除数据库。

--require_indexes

禁用在 index.yaml 文件中自动生成条目。当应用程序创建的查询要求在该文件中定义其索引，而未找到索引定义时，会抛出异常，这与在 App Engine 上运行时发生的情况类似。

--smtp_host=...

用于发送电子邮件的 SMTP 服务器主机名。

--smtp_port=...

用于发送电子邮件的 SMTP 服务器端口号。

--smtp_user=...

用于发送电子邮件的用户名（与 SMTP 服务器一同使用）。

--smtp_password=...

用于发送电子邮件的密码（与 SMTP 服务器一同使用）。

--enable_sendmail

使用本地计算机上安装的 Sendmail 发送电子邮件。

--debug_imports

打印与导入模块相关的调试消息（包含搜索路径和错误）。

上传应用程序

App Engine SDK 包括用于上传应用程序文件的命令。与 SDK 的其余部分一样，appcfg.py 可在任何装有 Python 2.5 的计算机上运行。

要上传应用程序文件，请使用 update 操作并输入应用程序根目录的名称来运行 appcfg.py 命令。根目录应包含应用程序的 app.yaml 文件。

appcfg.py update myapp/

appcfg.py 将从 app.yaml 文件获得应用程序 ID，并提示您输入 Google 帐户的电子邮件地址和密码。使用您的帐户成功登录后，appcfg.py 会存储一个 [cookie]，这样以后登录时就无需再提示输入密码。

您可以使用 --email 选项在命令行指定电子邮件地址。不能将密码指定为命令行选项。

appcfg.py --email=Albert.Johnson@example.com update myapp/

appcfg.py 选项

appcfg.py 命令包含一组选项、一个操作和该操作的参数。

有以下操作可选：

appcfg.py *[options]* update *<app-directory>*

为指定了应用程序根目录的应用程序上传文件。应用程序 ID 和版本是从应用程序目录中的 app.yaml 文件获取的。

appcfg.py *[options]* rollback *<app-directory>*

撤消指定应用程序完成的部分更新。如果更新中断并且命令报告由于锁定而无法更新应用程序，则可以使用此操作。

appcfg.py *[options]* vacuum_indexes *<app-directory>*

删除 App Engine 中不使用的数据库索引。如果从 index.yaml 中删除索引定义，则当上传应用程序时，该索引不会自动删除，因为它可能正被该应用程序的其他版本使用。当不再需要所有旧索引时执行此操作。

appcfg.py *[options]* request_logs *<app-directory>* *<output-file>*

检索 App Engine 上运行的应用程序的日志数据。*output-file* 是要创建、替换或附加（如果设置了 --append 标记）的文件的名称。如果 *output-file* 为连字符 (-)，则日志数据将打印到控制台。以下选项适用于 request_logs：

--num_days=...

要检索的日志数据的天数（结束时间为国际协调时间当前日期的午夜）。0 值会检索所有可用日志。如果 --append 已指定，则默认为 0，否则默认为 1。

--severity=...

要检索的日志消息的最低日志级别。该值是一个与日志级别相对应的数字：4 表示 [严重]，3 表示 [错误]，2 表示 [警告]，1 表示 [信息]，0 表示 [调试]。指定日志级别及更高级别的所有消息都会被检索。默认为 1（信息）。

--append

将抓取的数据附加到输出文件，开头为文件中尚未出现的第一个日志行。每天运行一次该命令，使文件中的 --append 结果包含所有日志数据。

默认为覆盖输出文件。如果 *output-file* 为 -（打印到控制台），则不适用。

appcfg.py help *<action>*

打印有关指定操作的帮助消息，然后退出。

appcfg.py 命令接受以下适用于所有操作的选项：

--quiet

成功时不打印消息。

--verbose

打印有关命令正在执行的操作的消息。

--noisy

打印有关命令正在执行的操作的多个消息。在与 App Engine 团队协作解决上传问题时，此选项最为有用。

--email=...

应用程序管理员的 Google 帐户的电子邮件地址，适用于需要登录的操作。如果忽略了该值且以前使用该命令时未存储 cookie，则该命令会提示输入。

--server=...

App Engine 服务器主机名。默认为 appengine.google.com。

--host=...

用于与远程过程调用配合使用的本地计算机的主机名。

--no_cookies

请勿将管理员登录凭证存储为 cookie；每次登录时都提示输入密码。

--force

强制删除不使用的索引。默认情况下，上传应用程序不会从服务器中删除不使用的索引，即使这些索引未显示在 index.yaml 文件中。

--max_size=...

要上传的文件的大小上限，以字节数表示。大于该大小的文件不会上传。默认为 1048576。服务器当前将文件的大小上限强制定为 1,048,576 字节，因此增加该值没有任何作用。

appcfg.py 和 HTTP 代理

如果您在 HTTP 代理下运行 appcfg.py，则必须告知 appcfy.py 该代理的名称。要为 appcfg.py 设置 HTTP 代理，请设置 http_proxy 环境变量。例如，在 bash shell 中：

```
export http_proxy="http://cache.mycompany.com:3128"

appcfg.py update myapp
```

管理控制台

Google App Engine 管理控制台为您提供可完整访问您的应用程序的公共版本的权限。通过在网络浏览器中访问以下链接来访问该控制台：

http://appengine.google.com/

使用您的 Google 帐户登录，或使用电子邮件地址和密码创建一个新的 Google 帐户。

如果您正通过您的 Google 企业应用套件帐户使用 App Engine，您可以使用类似下面的网址登录至您的域的 App Engine，其中 *your-domain.com* 是您的 Google 企业应用套件域：

http://appengine.google.com/a/*your-domain.com*

您可以使用管理控制台执行以下操作：

- 创建新应用程序，并设置一个免费的 appspot.com 子域或您选择的顶级域名
- 邀请其他人员成为您的应用程序的开发人员，这样，这些人员就可以访问该控制台并上传代码的新版本
- 查看访问数据和错误日志，并分析访问量
- 浏览您的应用程序的数据库并管理索引
- 测试您的应用程序的新版本，并切换用户看到的版本

有关使用 appcfg.py 命令上传您的应用程序代码的信息，请参阅上传应用程序。