**附件2：外文原文（复印件）**

Web Applications

This chapter focuses on the actual act of programming—on what it means to sit down and write a Python web application. Every other issue that we consider will be in the service of this overarching goal: to create a new web service using Python as our language. The work of designing a web site can be enormous and incur months of graphic design and usability work. Or it can involve nothing more than a single-page sketch on the back of a napkin. It can even be as simple as an idea in your head. But when it comes to implementation, applications that are designed to face the public Internet demand at least three big decisions from their implementers:

• A front-end web server will need to be chosen. Its job is to listen on port 80 of the web server—or whatever port has been designated for the site—and to serve static content like images, style sheets, and JavaScript files. And, for the specific URLs that serve the actual dynamic site content, the front-end server needs to delegate page creation to your Python program.

• Some means of linking the server and the Python application needs to be selected. We will spend the most time on the WSGI standard, which provides a standard invocation protocol between a web server and Python; however, it is also common for servers and Python to be linked through mechanisms like FastCGI and SCGI.

• Either in the web server itself or in the harness that runs the Python code, there needs to be logic that spawns several copies of the Python web application code, whether as threads or processes. This enables your app to answer different customers simultaneously without blocking.

• Finally, the programmer needs to decide which Python libraries he will use for common tasks like URL dispatch, database access, and template rendering—or whether to do without the convenience of standard tools altogether and to roll some of these solutions on his own.

Very often, the process of building a web application goes through these bullet points in reverse order. Most often, a programmer starts experimenting with an idea by running the “create project” routine of a popular web framework and adding her own code to the skeleton that gets created. Days or weeks or months later, when it is time to start exposing her application to real users on the local intranet or even out on the World Wide Web, the developer belatedly researches the best choice of front-end server for her framework of choice. She spends a few hours getting everything tweaked and configured correctly, so she can put her application into production. But we will tackle the steps in the order listed previously, moving from the front end of the system towards its core. This means that we will first establish the context in which Python web services run, and then spend the rest of the chapter focusing on actual programming techniques.

Acceptable web site performance generally requires the ability to serve several users concurrently. And since few Python programmers condescend to writing their web application logic using Twisted callbacks (see Chapter 7), achieving this performance means running several copies of your web application concurrently, using either threads or processes.

You will recall from our discussion of threads in Chapter 7 that the standard C language implementation of Python—the version of Python people download from its web site—does not actually run Python code in a thread-safe manner. To avoid corrupting in-memory data structures, C Python employs a Global Interpreter Lock (GIL), so that only one thread in a multi-threaded program can actually be executing Python code at any given time. Thus Python will let you create as many threads as you want in a given process; however, only one thread can run code at a time, as though your threads were confined to a single processor.

You might think that multiprocessing would always be the required approach; however, it turns out that threading can have decent performance because so many web applications are essentially light front-ends that sit between the user and a database. A typical web application receives and parses the user's request, then makes a corresponding request to the database behind it; while that thread is waiting for a response from the database, the GIL is available for any other threads that need to run Python code. Finally the database answers; the waiting thread reacquires the GIL; and, in a quick blaze of CPU activity, the data is turned into an attractive web page, and the response is sent winging its way back to the user.

Thus threads can sometimes at least perform decently. Nevertheless, multiple processes are the more general way to scale. This is because, as a service gets bigger, additional processes can be brought up on additional machines, rather than being confined to a single machine. Threads, no matter their other merits, cannot do that!

There are two general approaches to running a Python web application inside of a collector of identical worker processes:

• The Apache web server can be combined with the popular mod\_wsgi module to host a separate Python interpreter in every Apache worker process.

• The web application can be run inside of either the flup server or the uWSGI server. Both of these servers will manage a pool of worker processes where each process hosts a Python interpreter running your application. The front-end web server can submit requests to flup using either the standard Fast CGI (FCGI) or Simple CGI (SCGI) protocol, while it has to speak to uWSGI in its own special “uwsgi” protocol (whose name is all lowercase to distinguish it from the name of the server).

Note that both approaches insist that a powerful, secure, name-brand web server face the actual customer, with your Python web application sitting safely behind it. This lets the web server use its fast, compiled code to reject obviously malformed or nonsensical HTTP requests, passing along to your application only those requests that are at least superficially parsable. It can also have performance benefits, as you will see in the next section.

When the front-end web server receives an HTTP request, consults the patterns listed in its configuration, and decides that this particular URL needs to be handled by a Python web application, how does it actually invoke the Python code? And how can that code then communicate back to the server, whether to signal an error, make a redirect, or return a particular block of data as the web page?

Integrating Python with web servers used to be the Wild West: every server presented programmers with different data formats and calling conventions. Small web programs written against one server's API would need to be ported before they could be used with another brand of web server; and web frameworks themselves had to maintain a separate entry point for each server which developers might want to use to deploy their applications.

This situation was much improved by the creation of PEP 333, which defines the Python Web Server Gateway Interface (WSGI)

WSGI introduced a single calling convention that every web server could implement, thereby making that web server instantly compatible with all of the Python web applications and web frameworks that also support WSGI.