urllib2 官方文档解析

20.6. urllib2 - 为打开URL的拓展库

注: urllib2 模块在Python 3 中被拆分成了 **urllib.request** 和 **urllib.error** . 当你将你的代码转换到 Python 3 时, *2to3* 工具能够自动地帮助你调整导入的模块

urllib2 模块定义了在复杂的世界中帮助你打开 URL (大部分是 HTTP)的函数和类 - 包括基本和摘要式认证,重定向,cookie和更多的一些功能.

1. urllib2 模块定义了以下函数:

- **urllib2.urlopen(url[, data][, timeout])**

打开参数 *url* 中的URL, *url* 参数可以是字符串类型或者 **Request** 对象.
警告: HTTPS 请求不会验证任何服务器的证书.
data 参数可能是指定发送到服务器的额外数据的字符串,如果没有这样的数据的话它将是 `None` .目前只有 HTTP 的请求会使用 *data* 参数: 当提供 *data* 参数时, HTTP 请求会将 GET 请求替换为 POST 请求. *data* 参数应当是标准的 *application/x-www-form-urlencoded* 格式的缓冲. **urllib.urlencode()** 函数接受一个映射或者包含二元元组的序列作为参数并返回一个符合格式的字符串. urllib2 模块发送包含了 `Connection:close` 头的 HTTP/1.1 请求.
可选参数 *timeout* 指定了类似于连接尝试的阻塞操作超时的秒数(如果没有指定的话,将使用全局的超时设置).这个参数仅对 HTTP , HTTPS 和 FTP 连接有效.
这个函数返回一个添加了以下三个方法的类文件对象:
- **geturl()** - 返回检索到的资源的URL,通常用于确定是否跟随了一个重定向

- **info()** - 以一个 mlmetools.Message 实例的形式返回页面的元信息,例如头

- **getcode()** - 返回 HTTP 响应的状态码
触发 URLError 的错误.
注意如果没有处理器处理请求(尽管默认情况下安装全局的 **OpenerDirector** 使用 **UnknownHandler** 来确保这种情况永远不会发生),函数可能返回None.
另外,如果检测到代理设置(例如,当一个 `*_proxy` 环境变量被设置时,比如 **http_proxy**), **ProxyHandler** 会被默认安装确保处理请求使之通过代理.

- **urllib2.install_opener(opener)**

安装一个 **OpenerDirector** 实例作为默认的全局 opener.安装一个 opener 仅在你希望 **urlopen()** 函数使用它时才是必须的;否则,简单的调用 **OpenDirector.open()** 来代替 **urlopen()** .代码不会检查一个真正的 **OpenerDirector** , 并且任何类相应的接口都将工作.

- **urllib2.build_opener([handler, ...])**

返回一个连接了按顺序给出的处理器的 **Openerdirector** 实例. *handlers* 可以是 **BaseHandler** 的任何实例,或者是 **BaseHandler** 的子类(在这种情况下它必须能够调用没有任何参数的结构).以下类的实例必须在 *handler* 之前,除非 *handlers* 包含了它们的实例或者他们的子类的实例: **ProxyHandler** (如果代理设置被检测到), **UnknownHandler** , **HTTPHandler** , **HTTPDefaultErrorHandler** , **HTTPRedirectHandler** , **FTPHandler** , **FileHandler** , **HTTPErrorProcessor**.
如果安装的 Python 支持 SSL (例如 **ssl** 模块能够被导入), **HTTPHandler** 也将被添加.
从 Python 2.3 开始,一个 **BaseHandler** 的子类也有可能改变它的 **handler_order** 属性以修改它在处理器列表中的位置.
以下异常会在会在合适的情况下触发:

- *exception* urllib2.####URLError####

处理器触发这个错误(或者衍生的错误)当它们运行时出了一个问题.它是 **IOError** 的子类.
- ####reason####
这个错误返回的原因可能是一个信息字符串或者另外一个异常实例(对远程 URL 的 **socket.error**,对本地 URL 的 **OSError**).

- *exception* urllib2.####HTTPError###

尽管是一个异常(**URLError** 的一个子类),一个 **HTTPError** 也可以作为一个函数无异常的类文件返回值(与 **urlopen** 返回的东西一样).这在处理特殊的 HTTP 错误时会非常有用,例如身份验证的请求
- ####code####
一个于 **RFC 2616** 标准中定义的 HTTP 状态码.这个数字值是 **BaseHTTPServer.BaseHTTPRequestHandler.responses** 状态码字典中对应的一个值
- ####reason####
这个错误返回的原因可能是一个信息字符串或者另外一个异常实例

2. urllib2 模块提供了以下类:

- **class urllib2.Request(url[, data][, headers][, origin_req_host][, unverifiable])**

这个类是对 URL 请求的抽象.
url 应该是一个包含有效 URL 的字符串.
data 参数可能是指定发送到服务器的额外数据的字符串,如果没有这样的数据的话它将是 `None` .目前只有 HTTP 的请求会使用 *data* 参数;当提供 *data* 参数时, HTTP 请求会将 GET 请求替换为 POST 请求. *data* 参数应当是标准的 *application/x-www-form-urlencoded* 格式的缓冲. **urllib2.urlencode()** 函数接受一个映射或者包含二元元组的序列作为参数并返回一个符合格式的字符串.
headers 应当是一个字典,可以看作是将每个键和值当作参数调用 **add_header()** 函数. 这经常用来修改浏览器用来声明自身的 `User-Agent` 头 - 一些常见HTTP服务器只允许请求来自浏览器而不是脚本.例如,火狐浏览器将声明自己为 `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"` ,同时 **urllib2** 的默认用户代理字符串为 `"Python-urllib/2.6"` (在 Python 2.6 中)
最后两个参数仅对正确地处理第三方 HTTP cookies 感兴趣:
origin_req_host 应当是在 **RFC 2965** 中定义的起源事务的请求主机.它默认为 `cookielib.request_host(self)` .这是被用户发起的主机名或 IP 地址.例如,如果请求是从一份 HTML 文档中获得一张图像,参数就应该是获得包含图像的页面的请求的请求主机.
unverfiable 应当声明请求是否是不被验证的,这在 **RFC 2965** 中被定义.它默认为 `False` .一个不被验证的请求是用户未选择允许选项的 URL.例如,如果请求是从一份 HTML 文档中获得一张图像,并且用户未选择选项允许自动地抓取图像,参数便应该为 `True`.

- **class urllib2.OpenerDirector**

这个类通过连接起来的 **BasHandler** 打开 URL ,它管理这些处理器链并且能从错误中恢复.

- **class urllib2.BaseHandler**

这是所有注册过的处理器的基类 -仅仅负责处理注册的简单结构.

- **class urllib2.HTTPDefaultErrorHandler**

这是一个为 HTTP 错误响应定义的默认处理器;所有的响应都转换成了 **HTTPError** 异常.

- **class urllib2.HTTPRedirectHandler**

这是一个为处理重定向的类.

- **class urllib2.HTTPCookieProcessor([cookiejar])**

这是一个为处理 HTTP Cookie的类.

- **class urllib2.ProxyHandler([proxies])**

是请求通过代理.如果 *proxies* 参数被提供,它一定是一个协议名称映射到代理地址的字典.默认从环境变量 **_proxy** 读取到代理列表.如果没有设置代理环境变量,在 Windows 环境中,那么从注册表的网络设置部分读取代理设置,在Mac OS X环境中,从OS X系统配置框架检索代理信息.
若要关闭自动代理检测填入一个空字典即可.

- **class urllib2.HTTPPasswordMgr**

保持一个 (realm, uri) -> (user, password) 映射数据库

- **class urllib2.HTTPPasswordMgrWithDefaultRealm**

保持一个 (realm, uri) -> (user, password) 映射数据库. 一个为`None` 的域被认为是匹配所有的域,它将被搜索如果没有其他的域匹配的话.

- **class urllib2.AbstractBasicAuthHandler([password_mgr])**

这是一个帮助 HTTP 验证的混入类,包括远程主机验证和代理验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的;涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

> "混入类被定义为‘一种被设计为通过继承与其他类结合的类’，它给其他类提供可选择的接口或功能。从实现上讲，混入类要求多继承；混入类通常是抽象类，不能实例化。混入类的作用在于：它不仅可以提高功能的重用性，减少代码冗余；而且还可以使相关的‘行为’集中在一个类中，而不是分布到多个类中，减少了避免 所谓的‘代码分散’和‘代码交织’问题，提高可维护性。"

- **class urllib2.HTTPBasicAuthHandler([password_mgr])**

处理远程主机的身份验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的; 涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

- **class urllib2.ProxyBasicAuthHandler([password_mgr])**

处理代理的验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的;涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

- **class urllib2.AbstractDigestAuthHandler([password_mgr])**

这是一个帮助 HTTP 验证的混入类,包括远程主机验证和代理验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的;涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

- **class urllib2.HTTPDigestAuthHandler([password_mgr])**

处理远程主机的身份验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的;涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

- **class urllib2.ProxyDigestAuthHandler([password_mgr])**

处理代理的验证. *password_mgr* 参数如果被提供,应当是与 **HTTPPasswordMgr** 相兼容的; 涉及到 **HTTPPasswordMgr** 对象部分的接口信息必须被支持.

- **class urllib2.HTTPHandler**

一个用于处理 HTTP 地址打开的类.

- **class urllib2.HTTPSHandler**

一个用于处理 HTTPS 地址打开的类.

- **class urllib2.FileHandler**

打开本地文件.

- **class urllib2.FTPHandler**

打开 FTP 地址.

- **class urllib2.CacheFTPHandler**

打开 FTP 地址,保持打开的 FTP 连接缓存以减少延迟.

- **class urllib2.UnknownHandler**

一个用于处理未知地址的全面的类.

- **class urllib2.HTTPErrorProcessor**

处理 HTTP 响应错误.

3. 对象

- **Request 对象**

以下方法描述了 **Request** 对象的公共接口,因此所有接口必须在子类中被覆盖.

`- Request.add_data(data)`
将请求数据设置为 *data* .除了 HTTP 处理器,其他处理器会忽略这个函数 - 并且参数应当是一个字节字符串,并且请求会由 `GET` 变为 `POST`.

`- Request.get_method()`
返回一个指示 HTTP 请求方法的字符串.这仅对 HTTP 请求有意义,并且一般总是返回 `GET` 或者 `POST` .

`- Request.has_data()`
检查是否实例含有 `POST` 数据.

`- Request.get_data()`
返回实例的 `POST` 数据.

`- Request.add_header(key, val)`
添加额外的一个请求头,头通常会被除 HTTP 处理器之外的所有处理器在被添加进头列表发送至服务器的地方忽略掉.注意,不能有两个相同名称的头,并且后添加的头会覆盖掉之前有冲突的头.目前来讲,这并不是 HTTP 功能的缺陷,因为所有有意义的头在使用超过两次时会有一种方法(header-specific)只使用一个头获得相同的功能.

`- Request.add_unredirected_header(key, header)`
添加一个不会被添加到重定向后的请求的头.

`- Request.has_header(header)`
检查是否实例拥有参数中的头(检查普通的头和不被重定向的头).

`- Request.get_full_url()`
返回在构造函数中传递的 URL .

`- Request.get_type()`
返回 URL 的类型 - 也被称为 `scheme` (*注:应该是协议的意思,待测试*)

`- Request.get_host()`
返回建立连接的主机.

`- Request.get_selector()`
返回选择器 - URL 中发送到服务器中的部分.

`- Request.get_header(header_name, default=None)`
返回指定头的值.如果这个头没有出现将返回默认值.

`- Request.header_items()`
返回一个包含所有请求头的元组(header_name, header_value)列表

- Request.set_proxy(host, type)
连接代理服务器以准备请求. *host* 和 *type* 将代替这些实例,并且实例的选择器将会变成给予构造函数的原始 URL .
- Request.get_origin_req_host()
返回起源事务的请求主机,这是在 **RFC 2965**中被定义的. 可以查看 **Request** 构造函数的文档.
- Request.is_unverifiable()
检查是否请求时不被验证的,这是在 **RFC 2965**中被定义的.可以查看 **Request** 构造函数的文档.

- **OpenerDirector 对象**

OpenerDirector 对象有以下方法:
- OpenerDirector.add_handler(handler)
handler 应当是一个 **BaseHandler** 实例.以下方法会被寻找并被添加到可能的链中(注意 HTTP 错误是特殊的情况).
  - *`protocol_open`* - 信号处理程序知道如何打开 `*protocol_open*` 地址.

  - *`http_error_type`* - 信号处理程序知道如何处理 HTTP 错误码类型.

  - *`protocol_error`* - 信号处理程序知道如何处理来自于协议(non-`http`)的错误.

  - *`protocol_request`* - 信号处理程序知道如何预处理协议请求.

  - *`protocol_response`* - 信号处理程序知道如何预处理协议响应.

- OpenerDirector.open(url[, data][, timeout])
打开提供的 *url* (可能是一个请求对象或者字符串), *data* 可以忽略.给予参数,返回值和触发的异常是和 **urlopen()** (简单的调用一般的全局 **OpenerDirector** 对象中的 **open()** 方法)一样的.可选参数 *timeout* 指定了类似于连接尝试的阻塞操作超时的秒数(如果没有指定的话,将使用全局的超时设置).这个参数仅对 HTTP , HTTPS 和 FTP 连接有效.
- OpenerDirector.error(proto[, arg[, ...]])
处理一个提供的协议的错误.这将为提供的带有参数(指明协议详情)的协议调用以注册的错误处理器. HTTP 是一个特殊的例子,因为它使用 HTTP 响应码来决定特定的错误处理器;这里指的是处理器类中的 **http_error_*()** 方法.
返回值和触发的异常是和 **urlopen()** 一样的.
OpenerDirector 对象打开地址分为三步:
每个阶段的方法在哪里以何种顺序被调用取决于处理器实例的顺序.
  1. 每个带有命名类似于 *`protocol_request`* 方法的处理器都会调用这个方法来预处理请求.

  2. 每个带有命名类似于 *`protocol_open`* 方法的处理器都会调用这个方法来处理请求.当一个非 **None** (例如一个响应)值被返回或者出发一个异常时 (通常是 **URLError**),这个阶段将会结束.异常可以传播.
实际上,以上算法首先尝试过 **default_open()** 方法.如果所有的方法返回 **None**,算法会重复命名类似于 *protocol_open* 的方法.如果以上所有的方法都返回 **None**,算法将会重复命名为 **unknown_open()** 的方法.
注意,这些方法的实现可能涉及到调用父类 **OpenerDirector** 实例的 **open()** 和**error()** 方法。
  3. 每个带有命名类似于 *`protocol_response`* 方法的处理器都会调用这个方法来后处理响应.

- **BaseHandler 对象**

BaseHandler 对象提供了一些直接有效的方法,另外一些方法则是被派生类使用的.这些都是用于直接使用:
- BaseHandler.add_parent(director)
Add a director as parent.
- BaseHandler.close()
Remove any parents.
以下属性和方法仅被 **BaseHandler** 的派生类使用.
注意: 子类对 **protocol_request** 和 **protocol_response()** 方法的命名约定为 ***Processor**;其它所有的命名为 ***Handler**.
- BaseHandler.parent
一个有效的 **OpenerDirector** 对象,它可以被用作用一个不同的协议打开,或者处理错误.
- BaseHandler.default_open(req)
这个方法不是在 **BaseHandler** 中被定义的,但是子类必须重写它如果它们想匹配所有的地址的话.
这个方法如果执行将会被其父 **OpenerDirector** 调用.它的返回值应当和 **OpenerDirector** 中的 **open()** 描述的返回值一样,是一个类文件对象,或者是 None .它应当触发 **URLError** ,除非一个真正异常的事件发生了(**MemoryError** 不应该映射到 **URLError**).
这个方法将在所有的特殊的协议打开方法前被调用.
- BaseHandler.protocol_open(req)
("protocol" 应当被替换为协议名称.)
这个方法不是在 **BaseHandler** 中被定义的,但是子类必须重写它如果它们想处理特定协议的地址的话.
这个方法如果被定义,将会被其父 **OpenerDirector** 调用.它的返回值应当和 **OpenerDirector** 中的 **default_open()** 返回值一样.
- BaseHandler.unknown_open(req)
这个方法不是在 **BaseHandler** 中被定义的,但是子类必须重写它如果它们想匹配所有未指定处理器的地址的话.
这个方法如果执行将会被其父 **OpenerDirector** 调用.它的返回值应当和 **OpenerDirector** 中的 **default_open()** 描述的返回值一样.

- BaseHandler.http_error_default(req, fp, code, msg, hdrs)

这个方法不是在 **BaseHandler** 中被定义的,但是子类必须重写它如果它们打算提供全面的错误处理支持.当产生错误时 **OpenerDirector** 会自动调用它,并且在其他正常环境下不应被调用.

req 是一个 **Request** 对象, *fp* 是一个带有 HTTP 错误体的类文件对象, *code* 是三位数错误码, *msg* 是用户可见的错误码解释, *hdrs* 是带有头错误信息的映射对象.

返回值和异常应当和 **urlopen()** 一致.

- BaseHandler.http_error_nnn(req, fp, code, msg, hdrs)

nnn 三位 HTTP 错误码.这个方法不是在 **BaseHandler** 中被定义的,当它存在于一个子类实例中并且一个对应的 HTTP 错误码出现时它便会被调用.

子类必须重写这个方法使之能够处理特定的 HTTP 错误.

参数,返回值和触发的异常应当和 **http_error_default()** 一致.

- BaseHandler.protocol_request(req)

("protocol" 应当被替换为协议名称.)

这个方法不是在 **BaseHandler** 中被定义的,但是子类必须定义它如果它们想预处理给定协议的请求.

这个方法如果被定义,将会被其父 **OpenerDirector** 调用. *req* 是一个 **Request** 对象.返回值应当是一个 **Request** 对象.

- BaseHandler.protocol_response(req, response)

("protocol" 应当被替换为协议名称.)

这个方法不是在 **BaseHandler** 中被定义的,但是子类必须定义它如果它们想预处理给定协议的响应.是一个对象,这个对象应当实现和 **urlopen()** 返回值一样的接口.

- **HTTPRedirectHandler 对象**

注意: 一些 HTTP 重定向需要模块客户端代码的操作.如果是这种情况,将会触发 **HTTPError** .查看 **RFC 2616** 可以获得各种重定向代码的具体含义.

  - HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)

返回一个 **Request** 或者 `None` 来响应重定向.当从服务器收到重定向时, ***http_error_30() 的默认实现会调用这个方法.如果要发生一个重定向,将会返回一个新的 Request 以允许 http_error_30*()**

执行重定向到新的地址.另外,如果没有其它处理器尝试处理这个地址,就会触发 **HTTPError** ,或者返回 `None` 你不得不使用一个可能的另外的处理器[*if no other handler should try to handle this URL, or return None if you can't but another handler might*].

注意:这个方法的默认实现并未严格按照 **RFC 2616** 标准,也就是说301和302响应对 `POST` 请求禁止在未经用户确认的情况下自动重定向.实际上,浏览器会自动允许这些重定向响应,并将 `POST` 改为`GET` ,默认实现再现了这种行为.

  - HTTPRedirectHandler.http_error_301(req, fp, code, msg, hdrs)

重定向到 `Location:` 或者 `URI:` 地址.当收到 HTTP 'moved permanently' 响应时这个方法被其父 **OpenerDirector** 调用.

  - HTTPRedirectHandler.http_error_302(req, fp, code, msg, hdrs)

与 **http_error_301()** 一样,但是称为 'found' 响应.

  - HTTPRedirectHandler.http_error_303(req, fp, code, msg, hdrs)

与 **http_error_301()** 一样,但是称为 'see other' 响应.

  - HTTPRedirectHandler.http_error_307(req, fp, code, msg, hdrs)

与 **http_error_301()** 一样,但是称为 'temporary redirect' 响应.

- **HTTPCookieProcessor 对象**

HTTPCookieProcessor 有一个属性:

  - HTTPCookieProcessor.cookiejar

cookielib.CookieJar是被储存的cookie.

- **ProxyHandler 对象**

  - ProxyHandler.protocol_open(request)

ProxyHandler 有一个为每一个在构造函数中提供的代理字典中的代理工作的 *`protocol_open`* 函数,这个函数将通过调用 `request.set_proxy()` 修改请求使之通过代理,并且调用在处理链中的下一个处理器使协议生效[call the next handler in the chain to actually execute the protocol].

- **HTTPPasswordMgr 对象**

这些方法在 **HTTPPasswordMgr** 和 **HTTPPasswordMgrWithDefaultRealm** 对象可用.

  - HTTPPasswordMgr.add_password(realm, uri, user, passwd)

uri 可以是单个的 URI ,也可以是一个 URI 的队列. *realm* , *user* 和 *passwd* 必须字符串.This causes (user, passwd) to be used as authentication tokens when authentication for realm and a super-URI of any of the given URIs is given.

- HTTPPasswordMgr.find_user_password(realm, authuri)

获得给定域和 URI 的用户名/密码.如果没有匹配到,将会返回 `(None,None)` .
对于 **HTTPPasswordMgrWithDefaultRealm** 对象,如果在给定域匹配到结果的话将会在 `None` 域中寻找.

- **AbstractBasicAuthHandler 对象**
  - AbstractBasicAuthHandler.http_error_auth_reqed(authreq, host, req, headers)

通过获取用户名/密码对来处理验证请求,并且再次尝试请求. *authreq* 应当为包括域信息的请求的头名称, *host* 指明了验证的地址和路径, *req* 应当为(failed) **Request** 对象, *headers* 应当为错误头.
host 是一个授权(例如: `"python.org"`)或者是一个包含授权成分的 URI .无论如何,授权都不能包含用户信息成分(因此, `"python.org"` 和 `"python.org:80"` 是可以的, `"joe:password@python.org"` 则不可以).

- **HTTPBasicAuthHandler 对象**
  - HTTPBasicAuthHandler.http_error_401(req, fp, code, msg, hdrs)

如果可以的话,重复带有验证信息的请求.

- **ProxyBasicAuthHandler 对象**
  - ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)

如果可以的话,重复带有验证信息的请求.

- **AbstractDigestAuthHandler 对象**
  - AbstractDigestAuthHandler.http_error_auth_reqed(authreq, host, req, headers)

authreq 应当为包括域信息的请求的头名称, *host* 指明了验证的地址和路径, *req* 应当为(failed) **Request** 对象, *headers* 应当为错误头.

- **HTTPDigestAuthHandler 对象**
  - HTTPDigestAuthHandler.http_error_401(req, fp, code, msg, hdrs)

如果可以的话,重复带有验证信息的请求.

- **ProxyDigestAuthHandler 对象**
  - ProxyDigestAuthHandler.http_error_407(req, fp, code, msg, hdrs)

如果可以的话,重复带有验证信息的请求.

- **HTTPHandler 对象**
  - HTTPHandler.http_open(req)

发送 HTTP 请求,它可以是 GET 或者 POST,这取决于 `req.has_data()` .

- **HTTPSHandler 对象**
  - HTTPSHandler.https_open(req)

发送 HTTPS 请求,它可以是 GET 或者 POST,这取决于 `req.has_data()` .

- **FileHandler 对象**
  - FTPHandler.ftp_open(req)

打开 *req* 指定的 FTP 文件.总是使用空用户名和密码登录.

- **CacheFTPHandler 对象**

CacheFTPHandler 对象相对 **FTPHandler** 对象有两个额外的方法:

- CacheFTPHandler.setTimeout(t)

设置连接超时的秒数.

- CacheFTPHandler.setMaxConns(m)

设置最大的缓存连接数.

- **UnknownHandler 对象**
    - UnknownHandler.unknown_open()

触发一个 **URLError** 异常.

- **HTTPErrorProcessor 对象**
    - HTTPErrorProcessor.http_response()

处理 HTTP 错误请求.
对于200错误码,会立刻返回响应对象
对于非200错误码,它会简单地通过 **OpenerDirector.error()** 将工作转交给 *protocol_error_code* 处理方法.最终,如果没有其他处理器处理错误的话, **urllib2.HTTPDefaultErrorHandler** 将会触发一个 **HTTPError** .

    - HTTPErrorProcessor.https_response()

处理 HTTPS 错误请求.
它的行为与 **http_response()** 一致.

4. 例子

获取 python.org 主页并显示前100个字节的例子:

```
import urllib2
f = urllib2.urlopen('http://www.python.org/')
print f.read(100)
```

在这里我们发送一个数据包到 CGI 的标准输入并且读取它返回给我们的数据.注意例子仅仅在 Python 程序支持 SSL 的条件下才能正常工作.

```
import urllib2
req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
                      data='This data is passed to stdin of the CGI')
f = urllib2.urlopen(req)
print f.read()
```

上例示例 CGI 使用的代码:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' % data
```

使用基本的 HTTP 验证:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

build_opener() 提供了许多默认处理器,包括了一个 **ProxyHandler** .默认地, **ProxyHandler** 使用名称为 `<scheme>_proxy` 的环境变量, `<scheme>` 是相关的地址方案.例如, **http_proxy** 环境变量被读取以获得 HTTP 代理的地址.
这个例子替换了默认的 **ProxyHandler**,使用了一个带有编程支持的代理地址的对象,并且使用 **ProxyBasicAuthHandler** 添加了代理认证支持.

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib2.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

添加 HTTP 头:
在 **Request** 构造函数中使用 *headers* 参数,或者:

```
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
r = urllib2.urlopen(req)
```

OpenerDirector 自动地添加 *User-Agent* 头到每一个 **Request** 中.为了改变它:

```
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

另外要记住的是,当 **Request** 传递到 **urlopen()** (或者 **OpenerDirector.open()**)时,一些标准头(*Content-Length, Content-Type and Host*)也会被添加.