每个 **Web** 开发者都应该知道的关于 **URL** 编码的知识

本文首先阐述了人们关于统一资源定位符（URL）编码的普遍的误读，其后通过阐明HTTP场景下的URL encoding 来引出我们经常遇到的问题及其解决方案。本文并不特定于某类编程语言，我们在Java环境下阐释问题，最后从Web应用的多个层次描述如何解决URL编码的问题来结尾。

目录

简介

当我们每天上网冲浪时，有一些技术我们无时无刻不在面对。有数据本身（网页），数据的格式化，能够让我们获取数据的传输机制，以及让Web网络能够真正成为Web的基础及根本：从一页到另一页的链接。这些链接都是URL。

通用URL语法

我敢说每个人在其一生中至少见过一次URL。比如"http://www.google.com"，就是一个URL。一个URL是一个*统一资源定位器* ，事实上它指向了一个网页(大多数情况下)。实际上，自从1994年的第一版规范开始，URL就有了一个良好定义的结构。

我们能从"http://www.google.com" 这个URL中读出下列详细信息:

Part	Data
Scheme	http
Host address	www.google.com

如果我们看一个更复杂的URL，比如 "https://bob:bobby@www.lunatech.com:8080/file;p=1?q=2#third" 我们就能获取到下列信息:

Part	Data
Scheme	https
User	bob
Password	bobby
Host address	www.lunatech.com
Port	8080
Path	/file
Path parameters	p=1
Query parameters	q=2
Fragment	third

协议 (即scheme，如上面的*http*和*https* (安全HTTP)) 定义了URL中其余部分的结构。大多数互联网URL协议 拥有通用的开头，包括*用户，密码，主机名和端口*，后面才是每个协议具体的部分。这个通用的部分负责处理认证，同时它也有能力知道为了请求数据应该链接到哪儿。

HTTP URL语法

对于HTTP URL (使用*http* 或 *https* 协议)，URL的scheme描述部分定义了数据的路径（*path*），后面是可选的*query* 和 *fragment*。

path 部分看上去是一个分层的结构，类似于文件系统中文件夹和文件的分层结构。path由"/"字符开始，每一个文件夹由"/"分隔，最后是文件。例如"/photos/egypt/cairo/first.jpg"有四个路径片段（segment）："photos"、"egypt"、"cairo" 和 "first.jpg"，可以由此推出："first.jpg" 文件在文件夹"cairo"中，而"egypt" 文件夹位于web站点的根文件夹"photos"里面。

每一个*path片段* 可以有可选的 *path参数* (也叫 matrix参数)，这是在path片段的最后由";"开始的一些字符。每个参数名和值由"="字符分隔，像这

样："/file;p=1"，这定义了*path片段* "file"有一个 *path参数* "p"，其值为"1"。这些参数并不常用 — 这得清楚 — 但是它们确实是存在，而且从Yahoo RESTful API 文档我们能找到很好的理由去使用它们：

> Matrix参数可以让程序在GET请求中可以获取部分的数据集。参考数据集的分页。因为matrix参数可以跟任何数据集的URI格式的path片段，它们可以在内部的path片段中被使用。

在 *路径(path)*部分之后是 *查询 (query)*部分，它和 *路径* 之间由一个"?"隔开， *查询*部分包含了一个由"&"分隔开的参数列表，每一个参数由参数名称、"="号以及参数值组成。比如"/file?q=2"定义了一个 *查询参数* "q" ，它的值是"2"。这在提交 HTML表单时，或者当你使用诸如Google搜索等应用时， 用的非常多。

一个HTTP URL的最后部分是一个*段落(fragment)*部分，用来指向HTML文件中具体的某个部分，而不是整个HTML页面。比如说，当你点击链接时浏览器自动滚屏到某个部分而不是从页面最顶部开始展示，就说明你点击了一个拥有*段落*部分的URL。

URL 语法

http URL 方案最初由 RFC 1738 定义（实际上，在之前的 RFC 1630也有涉及），而在 *http* URL 方案被重新定义之前，整个 URL 语法就已经由扩展了几次 以适应发展的规范进化为一套 *统一资源标识符 (Uniform Resource Identifiers 即 URIs)*。

对于 URLs 如何拼装，各部分如何分隔有一套语法。例如："://"分隔*方案*和*主机*部分。*主机*同*路径片段*部分由"/"分隔，而*查询*部分紧跟在"?"之后。这意味着有些字符为语法*保留*。有些为整个URIs保留，而有些则被特定方案保留。所有出现在不应出现位置的 *保留*符（例如*路径片段*——以文件名为例——可能包含"?"）必须被*URL 编码*。

URL 编码将字符转变成对 URL 解析无意义的无害形式。它将字符转化成为一种特定字符编码的字节序列，然后将字节转换为16进制形式，并将其前面加上"%"。问号的 URL 编码形式为"%3F"。

我们可以将指向 "to_be_or_not_to_be?.jpg"图片的 URL 写成："http://example.com/to_be_or_not_to_be%3F.jpg"，这样就没有人会认为这儿可能由一个*查询*部分了。

现今多数浏览器*显示* URLs 前都会对其*解码*（将*百分号编码*字节转回其原本字符），并在获取其网络资源的时候重新编码。这样一来，很多用户从未意识到编码的存在。

另一方面，网页作者，开发者必须明确认识到这一点，因为这里存在着很多陷阱。

URL常见陷阱

如果你正和URL打交道，了解下能够避免的常见陷阱绝对是值得的。现在我们给大家介绍下不仅限于此的一些常见陷阱。

使用哪类字符编码?

URL编码规范并没有定义使用何种字符编码形式去编码字节。一般的ASCII字母数字字符并不需要转义，但是ASCII之外的保留字需要(例如法语单词"nœud"中的"œ")。我们必须提出疑问，应该使用哪类字符编码来编码URL字节。

当然如果只有Unicode的话，这个世界就会清净很多。因为每个字符都包含其中，但是它只是一个集合，或者说是列表如果你愿意，它本身并不是一中编码。Unicode可以使用多种方式进行编码，譬如UTF-8或者UTF-16（也有其它格式），但是问题并没有解决：我们应该使用哪类字符来编码URL（通常也指URI）。

标准并没有定义一个URI应该以何种方式指定其编码，所以其必须从环境信息中进行推导。对于HTTP URL，它可以是HTML页面的编码格式，或HTTP头的。这通常会让人迷惑，也是许多错误的根源。事实上，最新版的URI标准 定义了新的URI scheme将采用UTF-8，host（甚至已有的scheme）也使用UTF-8，这让我更加怀疑：难道host和path真的可以使用不同的编码方式?

每一部分的保留字都是不同。

是的，他们是，是的，他们，是的，他们是。。。

对于一个httpd连接，路径片段部分中的空格被编码为"%20"（不，完全没有"+"）,而"+"字符在路径片段部分可以保持不编码。

现在，在查询部分，一个空格可能会被编码为"+"（为了向后兼容：不要试图在URI标准去搜索他）或者"%20"，当作为"+"字符（作为个统配符的结果）会被编译为"%2B"。

这意味着"blue+light blue"字串，如果在路径部分或者查询部分，将会有不同的编码。比如得到"http://example.com/blue+light%20blue?blue%2Blight+blue"这样的编码形式，这样我们不需从语法上分析url结构，就可以推导这个url的整个结构是可能

考虑如下组装URL的Java代码片段

```
String str = "blue+light blue";
String url = "http://example.com/" + str + "?" + str;
```

编码URL并不是为了转义保留字而进行的简单字符迭代，我们需要确切的知道哪个URL部份有哪些保留字，而有针对性的进行编码。

这也意味着URL重写过滤器如果不考虑合适的编码细节而对URL直接进行分段转换通常是有问题的。对URL进行编码而不考虑具体的分段规则是不切实际的。

保留字不是你想象的那样

大多数人不知道"+"在路径部分是被允许的并且特指正号而不是空格。其他类似的有：

- "?"在查询部分允许不被转义，
- "/"在查询部分允许不被转义，
- "="在作为路径参数或者查询参数值以及在路径部分允许不被转义，
- ":@-._~!$&'()*+,;="等字符在路径部分允许不被转义，
- "/?:@-._~!$&'()*+,;="等字符在任何段中允许不被转义。

这样下面的地址虽然看起来有点混乱:"http://example.com/:@-._~!$&'()*+,=;:@-._~!$&'()*+,=:@-._~!$&'()*+,==?/?:@-._~!$'()*+,;=/?:@-._~!$'()*+,;==#/?:@-._

按照上面的规则，其实上是一个合法的地址。

不用奇怪，上面路径可以被解析为：

部分	值
协议	http
主机	example.com
路径	/:@-._~!$&'()*+,=
路径参数名	:@-._~!$&'()*+,
路径参数值	:@-._~!$&'()*+,==
查询参数名	/?:@-._~!$'()* ,;
查询参数值	/?:@-._~!$'()* ,;==
段	/?:@-._~!$&'()*+,;=

不能分析解码后的URL

URL的语法只在它被解码前是有意义的，一旦解码就可能出现保留字。

例如"http://example.com/blue%2Fred%3Fand+green" 在解码前由如下部分组成：

Part	Value
Scheme	http
Host	example.com
Path segment	blue%2Fred%3Fand+green
Decoded Path **segment**	blue/red?and+green

这样看来，我们是在请求一个名为"blue/red?and+green"的文件，而不是一个位于"blue"文件夹下的名为"red?and+green"的文件。

如果我们把它解码为"http://example.com/blue/red?and+green"，我们将得到如下部分：

Part	Value
Scheme	http
Host	example.com
Path segment	blue
Path segment	red
Query parameter name	and green

这明显是错误的，所以，对保留字和URL各部分的分析必须在URL解码之前完成。这意味着URL重写过滤器不应当在尝试匹配之前解码URL，当且仅当保留字允许进行URL编码时才可以(有时符合这种情形，有时不符合，这取决于你的应用)。

解码后的URL不能被再编码为同样的形式

如果你解码"http://example.com/blue%2Fred%3Fand+green" 为"http://example.com/blue/red?and+green"，然后对它进行编码(哪怕使用一个对URL每一部分都很了解的编码器)，你将会得到"http://example.com/blue/red?and+green"，这是因为它已经是一个*有效的*URL。它跟我们解码之前的URL非常的不同。

用Java正确处理URL

当你觉得自己已经拿到了URL的黑腰带(柔道中的最高级别--译者注)，你将会发现仍有一些Java里特有的、URL相关的陷阱。如果没有一个强大的心脏，你很难正确的处理URL。

不要用java.net.URLEncoder或者java.net.URLDecoder来处理整个URL

不开玩笑。这些类不是用来编码或解码URL的，API文档中清楚的写着：

> Utility class for HTML form encoding. This class contains static methods for converting a String to theapplication/x-www-form-urlencodedMIME format. For more information about HTML form encoding, consult the HTML specification.

这不是给URL用的。充其量它类似于*查询* 部分的编码方式。使用它来编码或解码整个URL是错误的。你肯定以为标准的JDK一定会有一个标准的类来正确的处理URL编码(是这样，只不过是各部分分开处理的)，但是要么是压根没有，要么是我们还没有发现。不过，这种臆测导致许多人错用了URLEncoder。

在对每一部分编码之前不要拼装URL

正如我们已经讲过的:完整构建后的URL不能再被编码。

以下面的代码为例：

```
1 String pathSegment = "a/b?c";
2 String url = "http://example.com/" + pathSegment;
```

如果"a/b?c" 是一个路径片段，那么不可能把"http://example.com/a/b?c" 转换回之前它的原样，因为它碰巧是一个有效的URL。之前我们已经解释过这一点。

下面是正确的代码：

```
1 String pathSegment = "a/b?c";
```

```
String url = "http://example.com/"
           + URLUtils.encodePathSegment(pathSegment);
```

这里我们使用了一个工具类URLUtils，它是我们自己开发的，因为网络上找不到一个详尽的足够快的工具类。上面的代码会带给你正确编码的URL "http://example.com/a%2Fb%3Fc"。

注意，同样的方式也适用于查询子串：

```
String value = "a&b==c";
String url = "http://example.com/?query=" + value;
```

这会给你"http://example.com/?query=a&b==c"，这是个有效的URL，而不是我们想得到的"http://example.com/?query=a%26b==c"。

不要期望 URI.getPath() 给你结构化的数据

因为一旦一个URL被解码，句法信息就会丢失，下面这样的代码就是错误的：

```
URI uri = new URI("http://example.com/a%2Fb%3Fc");
for(String pathSegment : uri.getPath().split("/"))
  System.err.println(pathSegment);
```

它会先将路径 "a%2Fb%3Fc"解码为 "a/b?c",然后在不应该分割的地方将地址分割为地址片段。

正确的代码使用的是 未解码的路径 ：

```
URI uri = new URI("http://example.com/a%2Fb%3Fc");

for(String pathSegment : uri.getRawPath().split("/"))
  System.err.println(URLUtils.decodePathSegment(pathSegment));
```

注意路径参数仍然存在：如果需要的话再处理它们。

不要期望 Apache Commons HTTPClient的URI类能够正确的做对

Apache Commons HTTPClient 3的 URI 类使用了Apache Commons Codec的URLCodec来做 URL编码, 正如 API文档提到的 它是有问题的，因为它犯了和使用java.net.URLEncoder同样的错误。它不但使用了错误的编码器，还错误的 按照每一部分都具有同样的预定设置进行解码。

在web应用的每一层修复URL编码问题

近来我们已经被动修复了许多应用中的URL编码问题。从在Java中支持它，到低层次的URL重写。这里我们会列出一些必要的修改。

总是在创建的时候进行URL编码

在我们的 HTML文件中，我们将所有出现：

```
var url = "#{vl:encodeURL(contextPath + '/view/' + resource.name)}";
```

的地方替换为：

```
var url = "#{contextPath}/view/#{vl:encodeURLPathSegment(resource.name)}";
```

查询参数也是类似的。

确保你的URL-rewrite过滤器正确的处理网址

Url 重写过滤器是一个重写过滤器，我们在seam中用于转化漂亮的地址去应用依赖的网址。

例如，我们用它把http://beta.visiblelogistics.com/view/resource/FOO/bar转化为http://beta.visiblelogistics.com/resources/details.seam?owner=FOO&name=bar。

很明显，这个过程包含了一些字符串从一个地址到另一个地址，这意味着我们要从路径部分解码并且把它重新编码为另一个查询值部分。

我们起初的规则，如下所示：

```
<urlrewrite decode-using="utf-8">
 <rule>
  <from>^/view/resource/(.*)/(.*)$</from>
  <to encode="false">/resources/details.seam?owner=$1&name=$2</to>
 </rule>
</urlrewrite>
```

从这我们可以看到在重写过滤器中只有两种方法处理网址重写：每一个的网址先被解码去做规则匹配（<to>模式），或者它不可用，所有规则去处理解码。在我们看来后者是比较好的选择，特别是当你要移动网址部分周围，或者想去包含URL解码路径分隔符的匹配路径部分时候。

在替换模式中（<to>模式）你可以使用内建的函数escape（String）和unescape（String）处理网站转码和解码。

在撰写这个文章的时候，Url Rewrite Filter Beta 3.2有一些bugs，限制住我们提高URL-correctness：

- 网址解码使用java.net.URLDecoder（这是错误的），
- escape(String)和unescape(String)内建函数使用java.net.URLDecoder和java.net.URLEncoder（不够强大，只能用于这个查询字串，所有的"&"或者"="不被转码）。

We therefore made a big patch fixing a few issues like URL decoding, and adding the inline functionsescapePathSegment(String)andunescapePathSegment(String).

我们因此做了一个大修正补丁，用于修正诸如网址解码问题以及增加内建函建escapePathSegment(String) 和 unescapePathSegment(String)

我们现在可以这样写，几乎不会有错误

```
```

```
<urlrewrite decode-using="null">
 <rule>
  <from>^/view/resource/(.*)/(.*)$</from>
  <-- Line breaks inserted for readability -->
  <to encode="false">/resources/details.seam
                     ?owner=${escape:${unescapePath:$1}}
                     &name=${escape:${unescapePath:$2}}</to>
 </rule>
</urlrewrite>
```

唯一可能出问题的地方是由于我们的补丁还不能解决以下的问题：

- 内建的escaping/unescaping函数应能只能编码，这已经做为下一个补丁（已经做完了），或者能从http请求来确定（还不支持），
- oldescape(String)和unescape(String)内建函数被保留了，并且仍然调用java.net.URLDecoder，而这个包在由于没有解决"&"和"="的问题，所以仍然有问题，
- 我需要增加更多的局部特定的编码和解码函数，
- 我们需要增加一个方法去鉴别per-rule解码行为，对照全局在<urlrewrite>。

我们一有时间，我们就会发布第二个补丁。

正确使用Apache mod-rewrite

Apache mod-rewrite是一个Apache Web服务器的网址重写模块。例如用它来把 http://beta.visiblelogistics.com/foo 的流量代理到http://our-internal-server:8080/vl/foo。

这是最后的要修正的事情，就像是Url Rewrite Filter，他默认解码网址给我们，并且从新编码重写过得网址给我们，这其实上是错误的，因为"解码的网址不能被重新编码"。

有一种方法可以避免这种行为，至少在我们的案例中我们没有转化一个网址部分到另一个网址，例如，我们不需要解码一个路径部分并且重新编码它到一个查询部分：没有加码也没有重编码。

我们通过THE_REQUEST来网址匹配来完成工作。他是完全的HTTP请求（包括HTTP方法和版本）联合解码。我们只要取host后面的URL部分，改变host和预设的/v/前缀和tada

```
...

# This is required if we want to allow URL-encoded slashes a path segment
AllowEncodedSlashes On

# Enable mod-rewrite
RewriteEngine on

# Use THE_REQUEST to not decode the URL, since we are not moving
# any URI part to another part so we do not need to decode/reencode

RewriteCond %{THE_REQUEST} "^[a-zA-Z]+ /(.*) HTTP/\d\.\d$" RewriteRule ^(.*)$ http://our-internal-server:8080/vl/%1 [P,L,NE]
```

结论

我希望阐明一些URL技巧和常见的错误。简而言之，能把它说明白就够了，但这不是一些人想象的那样简单的。我们展示了java常见的错误和一个web 应用部署的整个过程。现在每个读者都应该是一个URL专家了，并且我们希望不要在看见相关bugs再出现。请求SUN公司，请为URL encoding/decoding逐项的增加标准支持

本文地址：http://www.oschina.net/translate/what-every-web-developer-must-know-about-url-encoding
原文地址：http://blog.lunatech.com/2009/02/03/what-every-web-developer-must-know-about-url-encoding

---