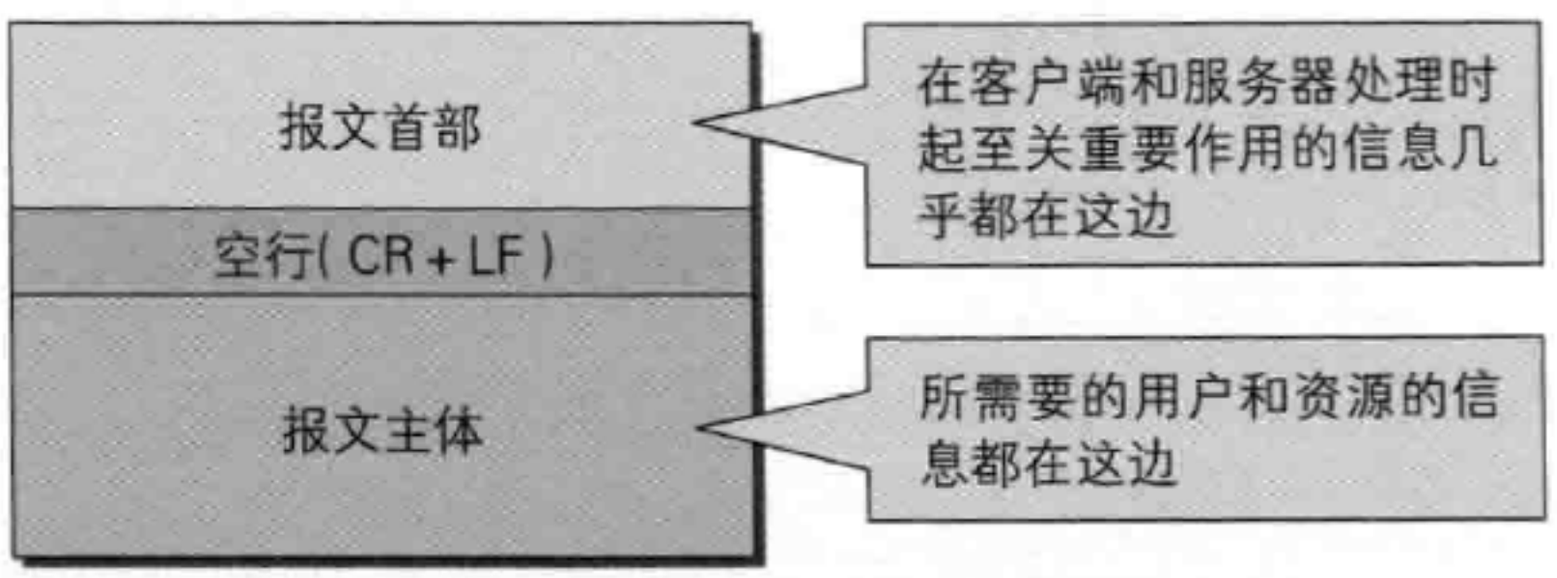
6.1 HTTP 报文首部



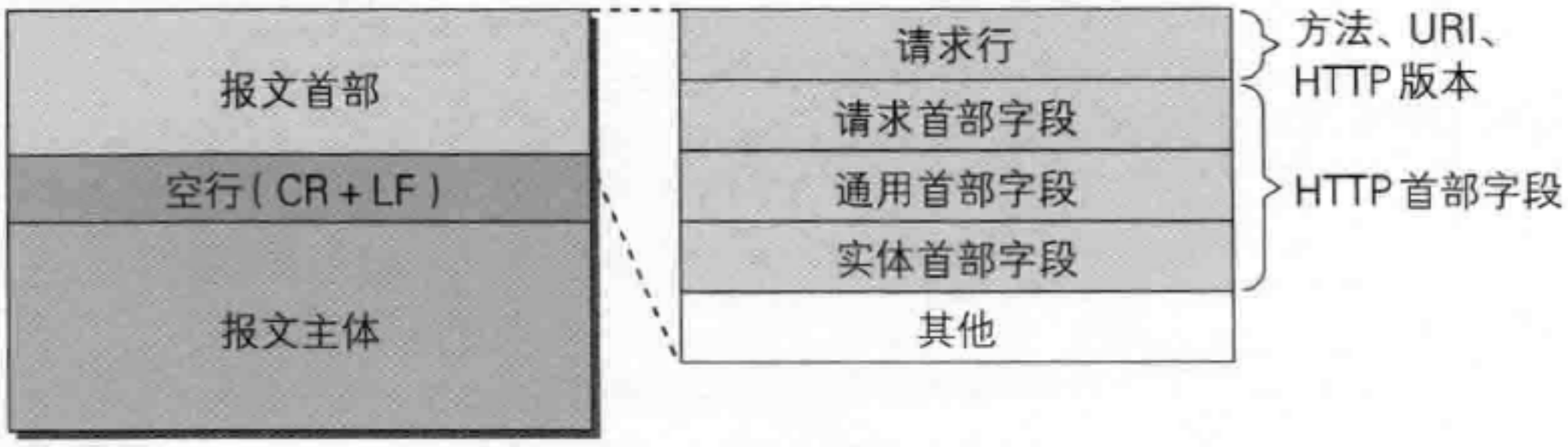
图：HTTP 报文的结构

HTTP 协议的请求和响应报文中必定包含 HTTP 首部。首部内容为客户端和服务器分别处理请求和响应提供所需要的信息。对于客户端用户来说，这些信息中的大部分内容都无须亲自查看。

报文首部由几个字段构成。

HTTP 请求报文

在请求中，HTTP 报文由方法、URI、HTTP 版本、HTTP 首部字段等部分构成。



图：请求报文

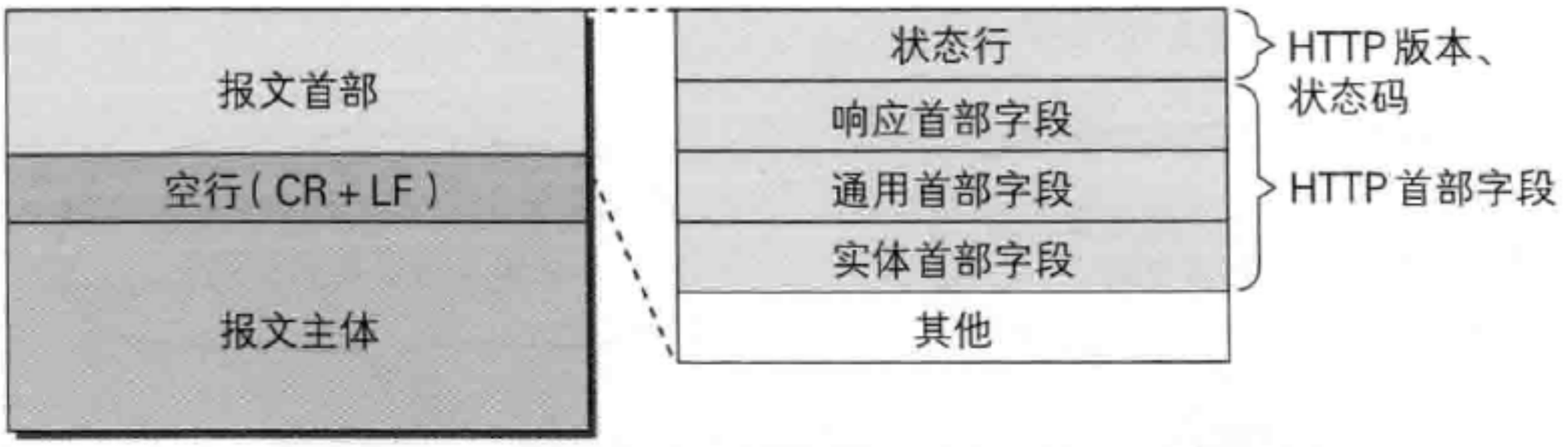
下面的示例是访问 http://hackr.jp 时，请求报文的首部信息。

```
GET / HTTP/1.1
Host: hackr.jp
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/⇒
20100101 Firefox/13.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,⇒
*/*; q=0.8
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
If-Modified-Since: Fri, 31 Aug 2007 02:02:20 GMT
If-None-Match: "45bae1-16a-46d776ac"
Cache-Control: max-age=0
```

HTTP 响应报文

在响应中，HTTP 报文由 HTTP 版本、状态码（数字和原因短语）、HTTP 首部字段 3 部分构成。



图：响应报文

以下示例是之前请求访问 http://hackr.jp/ 时，返回的响应报文的首部信息。

```
HTTP/1.1 304 Not Modified
Date: Thu, 07 Jun 2012 07:21:36 GMT
Server: Apache
Connection: close
Etag: "45bae1-16a-46d776ac"
```

在报文众多的字段当中，HTTP 首部字段包含的信息最为丰富。

首部字段同时存在于请求和响应报文内，并涵盖 HTTP 报文相关的内容信息。

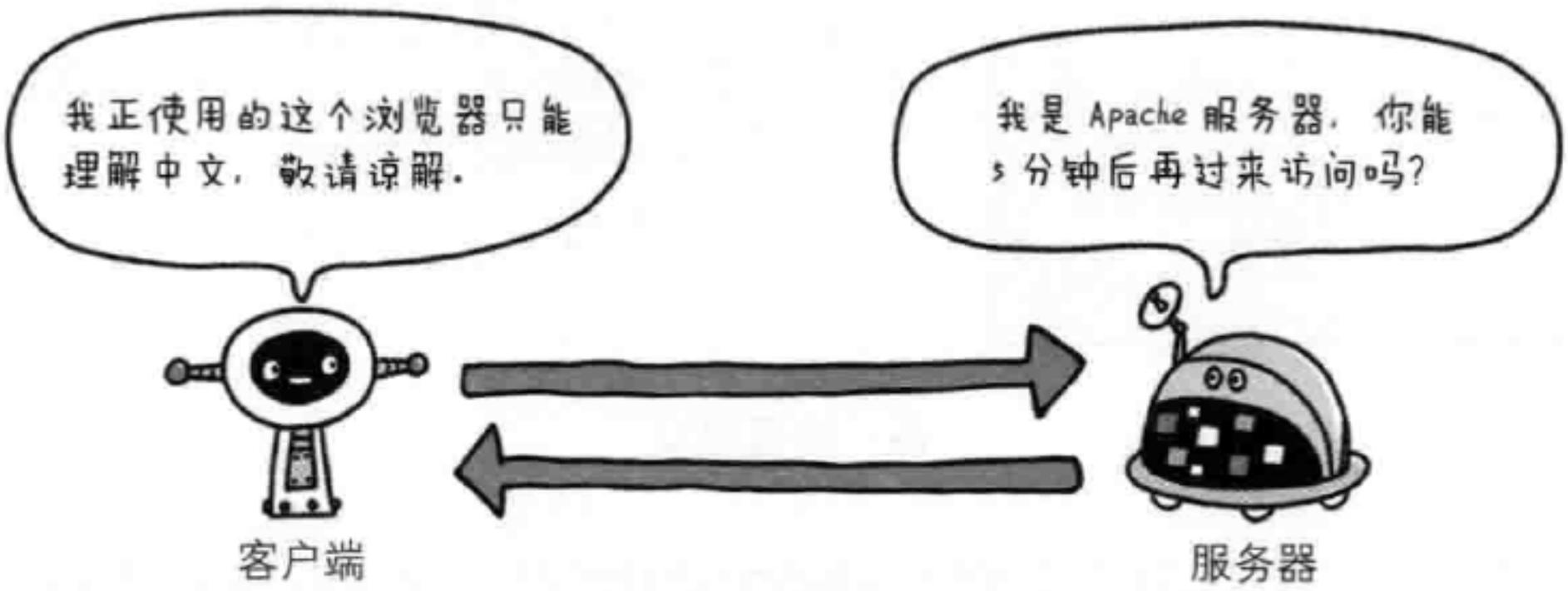
因 HTTP 版本或扩展规范的变化，首部字段可支持的字段内容略有不同。本书主要涉及 HTTP/1.1 及常用的首部字段。

6.2 HTTP 首部字段

6.2.1 HTTP 首部字段传递重要信息

HTTP 首部字段是构成 HTTP 报文的要素之一。在客户端与服务器之间以 HTTP 协议进行通信的过程中，无论是请求还是响应都会使用首部字段，它能起到传递额外重要信息的作用。

使用首部字段是为了给浏览器和服务器提供报文主体大小、所使用的语言、认证信息等内容。



图：首部字段内可使用的附加信息较多

6.2.2 HTTP 首部字段结构

HTTP 首部字段是由首部字段名和字段值构成的，中间用冒号“:”分隔。

```
首部字段名： 字段值
```

例如，在 HTTP 首部中以 Content-Type 这个字段来表示报文主体的对象类型。

```
Content-Type: text/html
```

就以上述示例来看，首部字段名为 Content-Type，字符串 text/html 是字段值。

另外，字段值对应单个 HTTP 首部字段可以有多个值，如下所示。

```
Keep-Alive: timeout=15, max=100
```


若 HTTP 首部字段重复了会如何

当 HTTP 报文首部中出现了两个或两个以上具有相同首部字段名时会怎么样？这种情况在规范内尚未明确，根据浏览器内部处理逻辑的不同，结果可能并不一致。有些浏览器会优先处理第一次出现的首部字段，而有些则会优先处理最后出现的首部字段。

6.2.3 4 种 HTTP 首部字段类型

HTTP 首部字段根据实际用途被分为以下 4 种类型。

通用首部字段（General Header Fields）

请求报文和响应报文两方都会使用的首部。



请求首部字段（Request Header Fields）

从客户端向服务器端发送请求报文时使用的首部。补充了请求的附加内容、客户端信息、响应内容相关优先级等信息。

响应首部字段（Response Header Fields）

从服务器端向客户端返回响应报文时使用的首部。补充了响应的附加内容，也会要求客户端附加额外的内容信息。

实体首部字段（Entity Header Fields）

针对请求报文和响应报文的实体部分使用的首部。补充了资源内容更新时间等与实体有关的信息。

6.2.4 HTTP/1.1 首部字段一览

HTTP/1.1 规范定义了如下 47 种首部字段。

表 6-1：通用首部字段

首部字段名	说明
Cache-Control	控制缓存的行为
Connection	逐跳首部、连接的管理
Date	创建报文的日期时间
Pragma	报文指令
Trailer	报文末端的首部一览
Transfer-Encoding	指定报文主体的传输编码方式
Upgrade	升级为其他协议
Via	代理服务器的相关信息
Warning	错误通知

表 6-2：请求首部字段

首部字段名	说明
Accept	用户代理可处理的媒体类型
Accept-Charset	优先的字符集
Accept-Encoding	优先的内容编码
Accept-Language	优先的语言（自然语言）
Authorization	Web认证信息
Expect	期待服务器的特定行为
From	用户的电子邮箱地址
Host	请求资源所在服务器
If-Match	比较实体标记（ETag）
If-Modified-Since	比较资源的更新时间
If-None-Match	比较实体标记（与If-Match相反）
If-Range	资源未更新时发送实体Byte的范围请求
If-Unmodified-Since	比较资源的更新时间（与If-Modified-Since相反）
Max-Forwards	最大传输逐跳数
Proxy-Authorization	代理服务器要求客户端的认证信息
Range	实体的字节范围请求
Referer	对请求中URI的原始获取方
TE	传输编码的优先级
User-Agent	HTTP 客户端程序的信息

表 6-3：响应首部字段

首部字段名	说明
Accept-Ranges	是否接受字节范围请求
Age	推算资源创建经过时间
ETag	资源的匹配信息
Location	令客户端重定向至指定URI

（续）

首部字段名	说明
Proxy-Authenticate	代理服务器对客户端的认证信息
Retry-After	对再次发起请求的时机要求
Server	HTTP服务器的安装信息
Vary	代理服务器缓存的管理信息
WWW-Authenticate	服务器对客户端的认证信息

表 6-4：实体首部字段

首部字段名	说明
Allow	资源可支持的HTTP方法
Content-Encoding	实体主体适用的编码方式
Content-Language	实体主体的自然语言
Content-Length	实体主体的大小（单位：字节）
Content-Location	替代对应资源的URI
Content-MD5	实体主体的报文摘要
Content-Range	实体主体的位置范围
Content-Type	实体主体的媒体类型
Expires	实体主体过期的日期时间
Last-Modified	资源的最后修改日期时间

6.2.5 非 HTTP/1.1 首部字段

在 HTTP 协议通信交互中使用到的首部字段，不限于 RFC2616 中定义的 47 种首部字段。还有 Cookie、Set-Cookie 和 Content-Disposition 等在其他 RFC 中定义的首部字段，它们的使用频率也很高。

这些非正式的首部字段统一归纳在 RFC4229 HTTP Header Field Registrations 中。

6.2.6 End-to-end 首部和 Hop-by-hop 首部

HTTP 首部字段将定义成缓存代理和非缓存代理的行为，分成 2 种类型。

端到端首部（End-to-end Header）

分在此类别中的首部会转发给请求 / 响应对应的最终接收目标，且必须保存在由缓存生成的响应中，另外规定它必须被转发。

逐跳首部（Hop-by-hop Header）

分在此类别中的首部只对单次转发有效，会因通过缓存或代理而不再转发。HTTP/1.1 和之后版本中，如果要使用 hop-by-hop 首部，需提供 Connection 首部字段。

下面列举了 HTTP/1.1 中的逐跳首部字段。除这 8 个首部字段之外，其他所有字段都属于端到端首部。

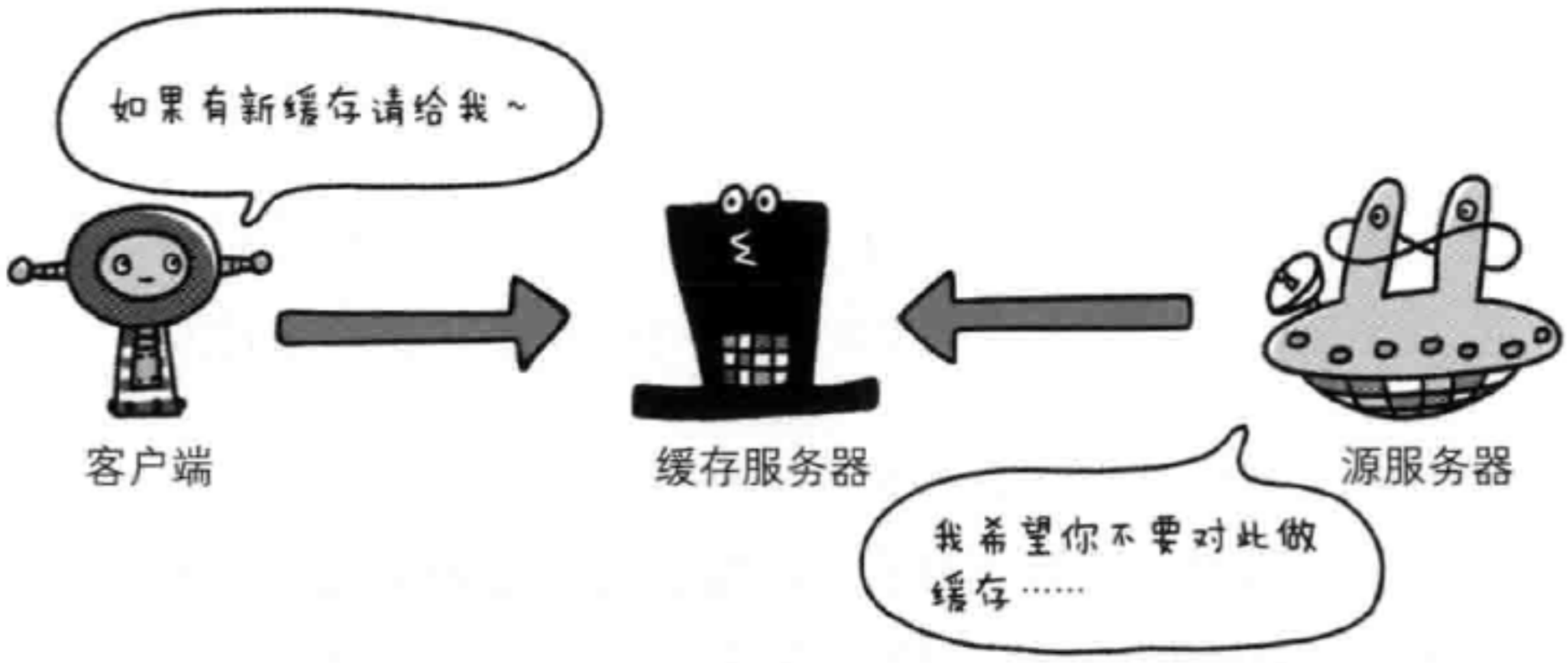
- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- Trailer
- TE
- Transfer-Encoding
- Upgrade

6.3 HTTP/1.1 通用首部字段

通用首部字段是指，请求报文和响应报文双方都会使用的首部。

6.3.1 Cache-Control

通过指定首部字段 Cache-Control 的指令，就能操作缓存的工作机制。



图：首部字段 Cache-Control 能够控制缓存的行为

指令的参数是可选的，多个指令之间通过“,”分隔。首部字段 Cache-Control 的指令可用于请求及响应时。


```
Cache-Control: private, max-age=0, no-cache
```

■ Cache-Control 指令一览

可用的指令按请求和响应分类如下所示。

表 6-5：缓存请求指令

指令	参数	说明
no-cache	无	强制向源服务器再次验证
no-store	无	不缓存请求或响应的任何内容
max-age = [秒]	必需	响应的最大Age值
max-stale(= [秒])	可省略	接收已过期的响应
min-fresh = [秒]	必需	期望在指定时间内的响应仍有效
no-transform	无	代理不可更改媒体类型

（续）

指令	参数	说明
only-if-cached	无	从缓存获取资源
cache-extension	-	新指令标记（token）

表 6-6：缓存响应指令

指令	参数	说明
public	无	可向任意方提供响应的缓存
private	可省略	仅向特定用户返回响应
no-cache	可省略	缓存前必须先确认其有效性
no-store	无	不缓存请求或响应的任何内容
no-transform	无	代理不可更改媒体类型
must-revalidate	无	可缓存但必须再向源服务器进行确认
proxy-revalidate	无	要求中间缓存服务器对缓存的响应有效性再进行确认
max-age = [秒]	必需	响应的最大Age值
s-maxage = [秒]	必需	公共缓存服务器响应的最大Age值
cache-extension	-	新指令标记（token）

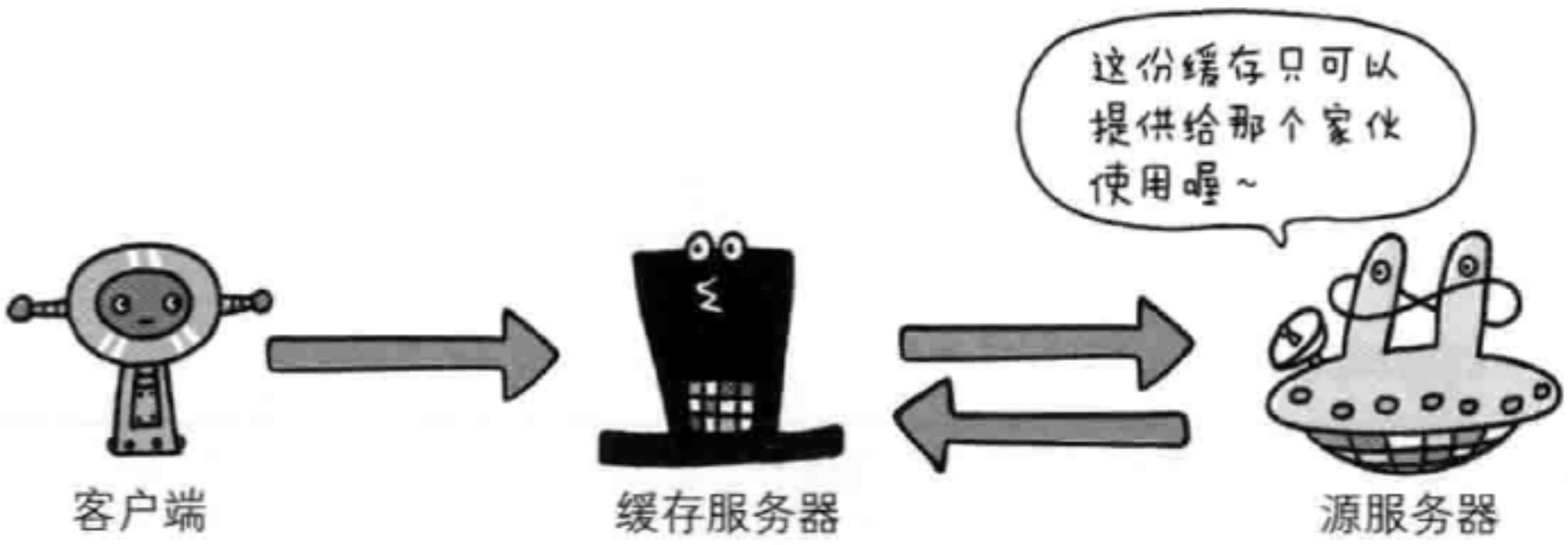
表示是否能缓存的指令

public指令

```
Cache-Control: public
```

当指定使用 public 指令时，则明确表明其他用户也可利用缓存。

private 指令

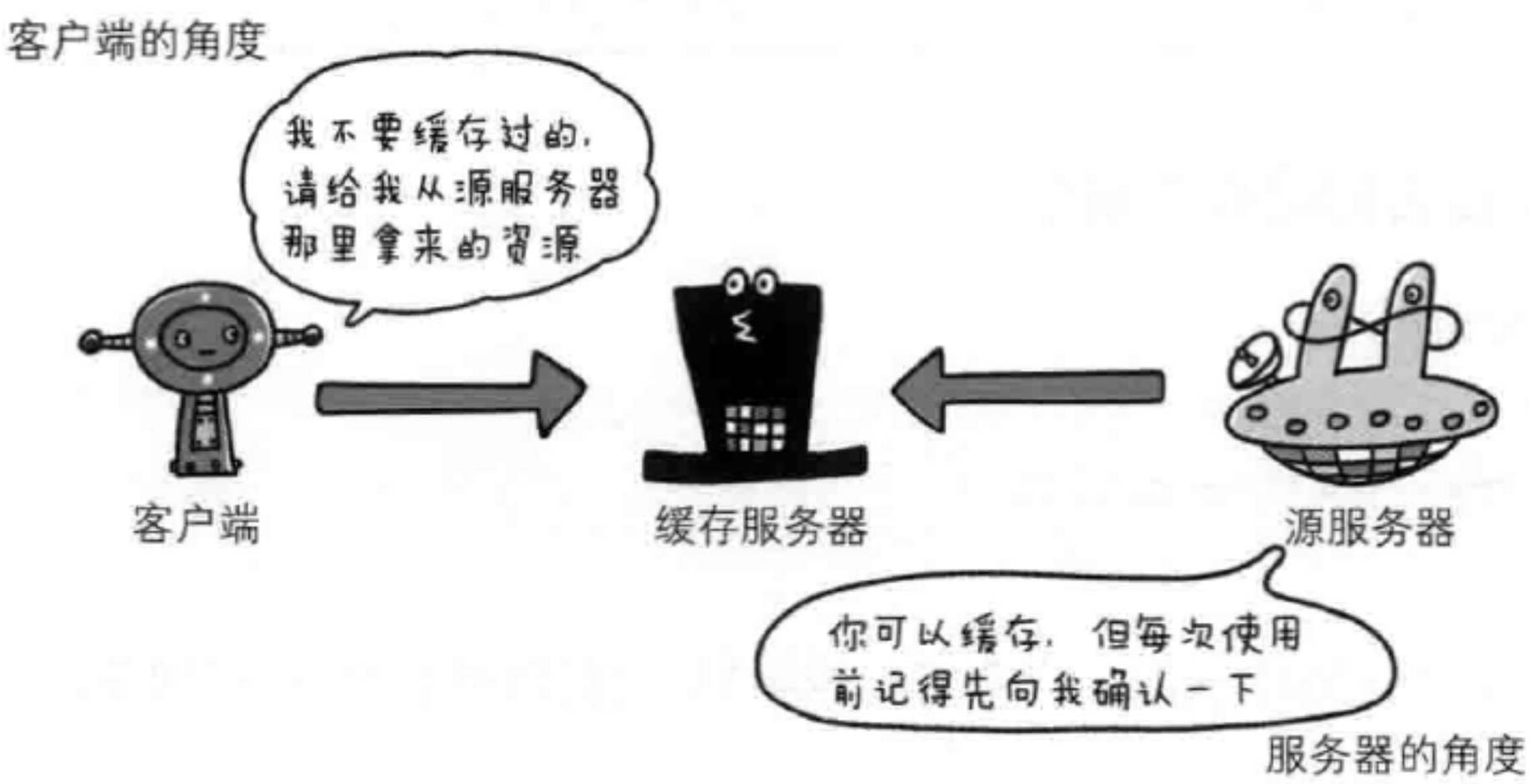



```
Cache-Control: private
```

当指定 private 指令后，响应只以特定的用户作为对象，这与 public 指令的行为相反。

缓存服务器会对该特定用户提供资源缓存的服务，对于其他用户发送过来的请求，代理服务器则不会返回缓存。

no-cache 指令




```
Cache-Control: no-cache
```

使用 no-cache 指令的目的是为了防止从缓存中返回过期的资源。

客户端发送的请求中如果包含 no-cache 指令，则表示客户端将不会接收缓存过的响应。于是，“中间”的缓存服务器必须把客户端请求转发给源服务器。

如果服务器返回的响应中包含 no-cache 指令，那么缓存服务器不能对资源进行缓存。源服务器以后也将不再对缓存服务器请求中提出的资源有效性进行确认，且禁止其对响应资源进行缓存操作。

```
Cache-Control: no-cache=Location
```

由服务器返回的响应中，若报文首部字段 Cache-Control 中对 no-cache 字段名具体指定参数值，那么客户端在接收到这个被指定参数值的首部字段对应的响应报文后，就不能使用缓存。换言之，无参数值的首部字段可以使用缓存。只能在响应指令中指定该参数。

控制可执行缓存的对象的指令

no-store 指令

```
Cache-Control: no-store
```

当使用 no-store 指令[①]时，暗示请求（和对应的响应）或响应中包含机密信息。

因此，该指令规定缓存不能在本地存储请求或响应的任一部分。

---

① 从字面意思上很容易把no-cache误解成为不缓存，但事实上no-cache代表不缓存过期的资源，缓存会向源服务器进行有效期确认后处理资源，也许称为do-not-serve-from-cache-without-revalidation更合适。no-store才是真正地不进行缓存，请读者注意区别理解。——译者注

指定缓存期限和认证的指令

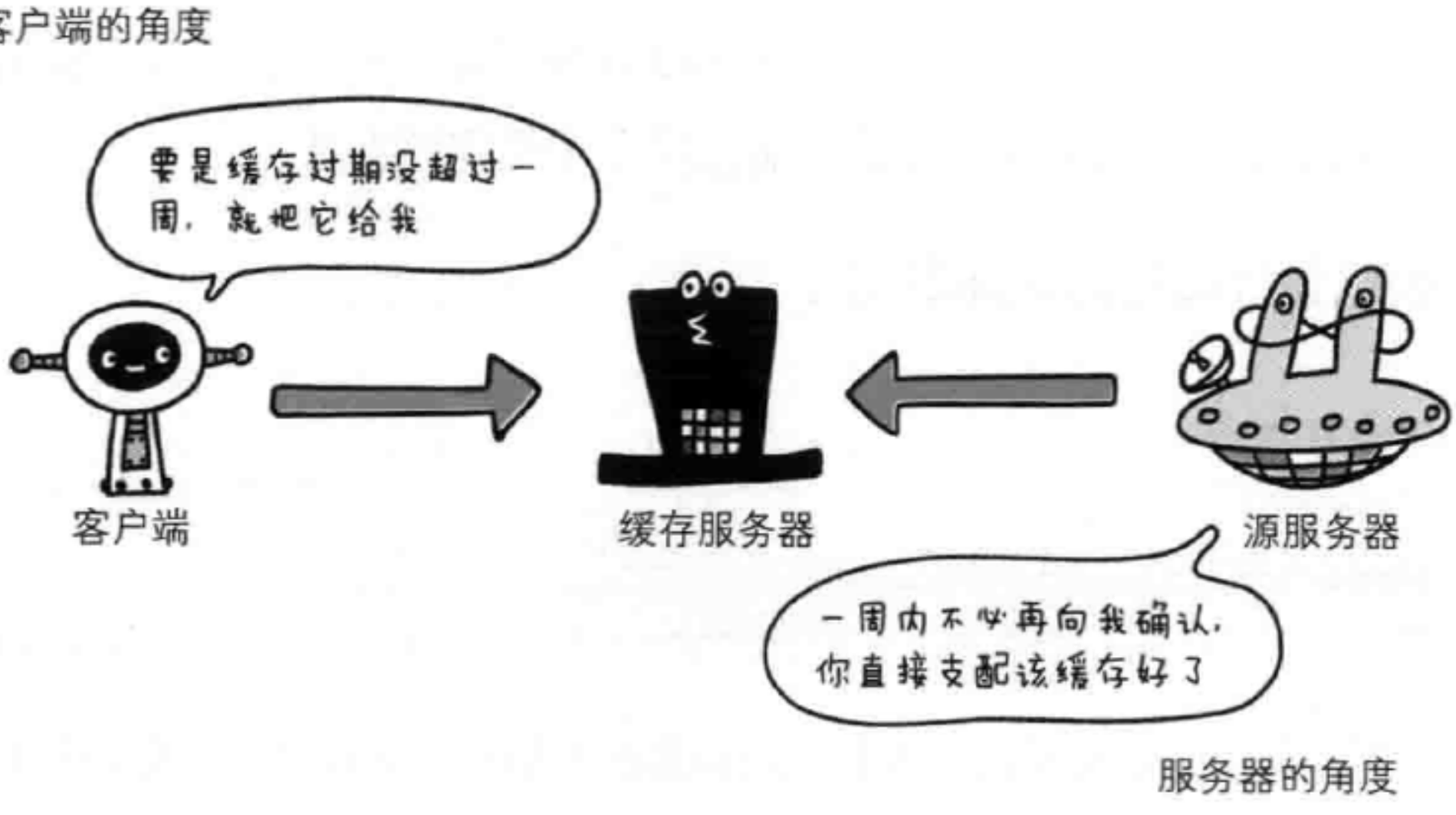
s-maxage指令

```
Cache-Control: s-maxage=604800（单位：秒）
```

s-maxage指令的功能和max-age指令的相同，它们的不同点是s-maxage指令只适用于供多位用户使用的公共缓存服务器[①]。也就是说，对于向同一用户重复返回响应的服务器来说，这个指令没有任何作用。

另外，当使用s-maxage指令后，则直接忽略对Expires首部字段及max-age指令的处理。

max-age指令




```
Cache-Control: max-age=604800（单位：秒）
```

当客户端发送的请求中包含max-age指令时，如果判定缓存资源的

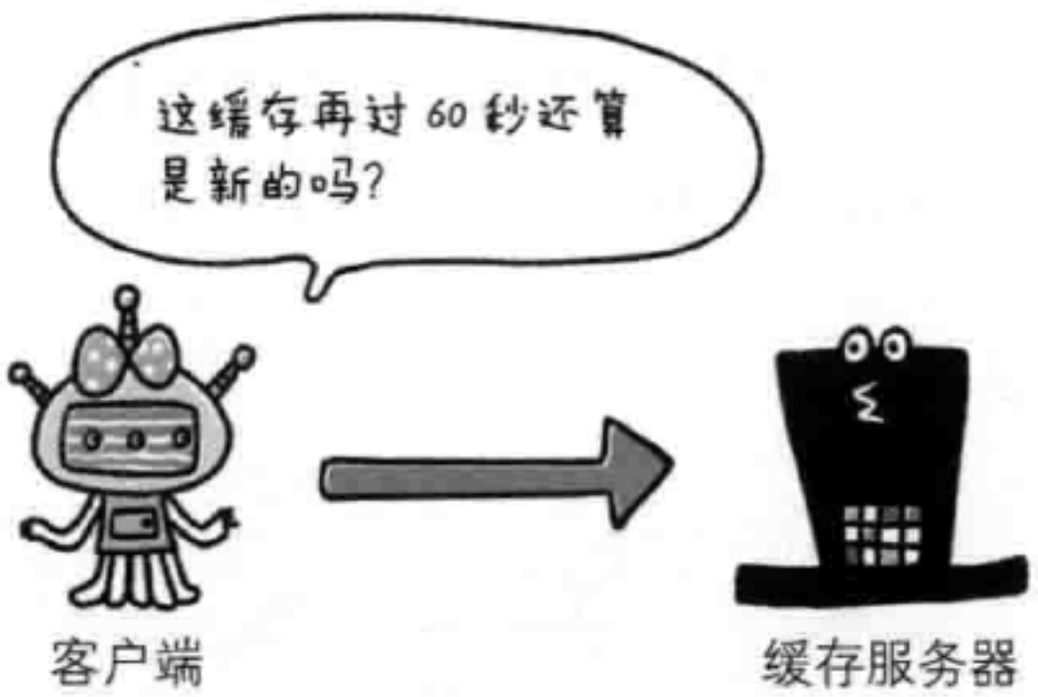
① 这里一般指代理。——译者注

缓存时间数值比指定时间的数值更小，那么客户端就接收缓存的资源。另外，当指定 max-age 值为 0，那么缓存服务器通常需要将请求转发给源服务器。

当服务器返回的响应中包含 max-age 指令时，缓存服务器将不对资源的有效性再作确认，而 max-age 数值代表资源保存为缓存的最长时间。

应用 HTTP/1.1 版本的缓存服务器遇到同时存在 Expires 首部字段的情况时，会优先处理 max-age 指令，而忽略掉 Expires 首部字段。而 HTTP/1.0 版本的缓存服务器的情况却相反，max-age 指令会被忽略掉。

min-fresh指令




```
Cache-Control: min-fresh=60（单位：秒）
```

min-fresh 指令要求缓存服务器返回至少还未过指定时间的缓存资源。

比如，当指定 min-fresh 为 60 秒后，过了 60 秒的资源都无法作为响应返回了。

max-stale指令

```
Cache-Control: max-stale=3600（单位：秒）
```

使用 max-stale 可指示缓存资源，即使过期也照常接收。

如果指令未指定参数值，那么无论经过多久，客户端都会接收响应；如果指令中指定了具体数值，那么即使过期，只要仍处于 max-stale 指定的时间内，仍旧会被客户端接收。

only-if-cached指令

```
Cache-Control: only-if-cached
```

使用 only-if-cached 指令表示客户端仅在缓存服务器本地缓存目标资源的情况下才会要求其返回。换言之，该指令要求缓存服务器不重新加载响应，也不会再次确认资源有效性。若发生请求缓存服务器的本地缓存无响应，则返回状态码 504 Gateway Timeout。

must-revalidate指令

```
Cache-Control: must-revalidate
```

使用 must-revalidate 指令，代理会向源服务器再次验证即将返回的响应缓存目前是否仍然有效。

若代理无法连通源服务器再次获取有效资源的话，缓存必须给客户端一条 504（Gateway Timeout）状态码。

另外，使用 must-revalidate 指令会忽略请求的 max-stale 指令（即使已经在首部使用了 max-stale，也不会再有效果）。

proxy-revalidate指令

```
Cache-Control: proxy-revalidate
```

proxy-revalidate 指令要求所有的缓存服务器在接收到客户端带有该指令的请求返回响应之前，必须再次验证缓存的有效性。

no-transform 指令

```
Cache-Control: no-transform
```

使用 no-transform 指令规定无论是在请求还是响应中，缓存都不能改变实体主体的媒体类型。

这样做可防止缓存或代理压缩图片等类似操作。

Cache-Control 扩展

cache-extension token

```
Cache-Control: private, community="UCI"
```

通过 cache-extension 标记（token），可以扩展 Cache-Control 首部字段内的指令。

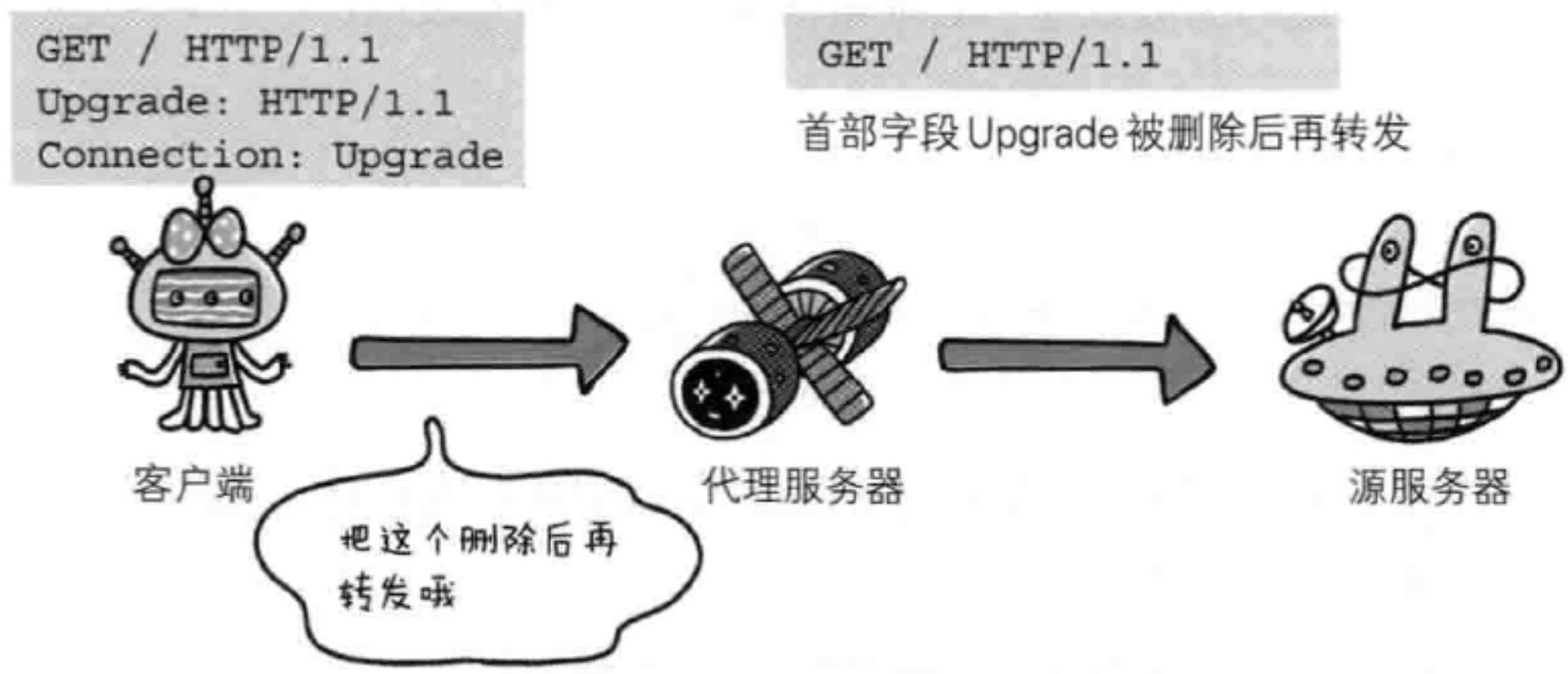
如上例，Cache-Control 首部字段本身没有 community 这个指令。借助 extension tokens 实现了该指令的添加。如果缓存服务器不能理解 community 这个新指令，就会直接忽略。因此，extension tokens 仅对能理解它的缓存服务器来说是有意义的。

6.3.2 Connection

Connection 首部字段具备如下两个作用。

- 控制不再转发给代理的首部字段
- 管理持久连接

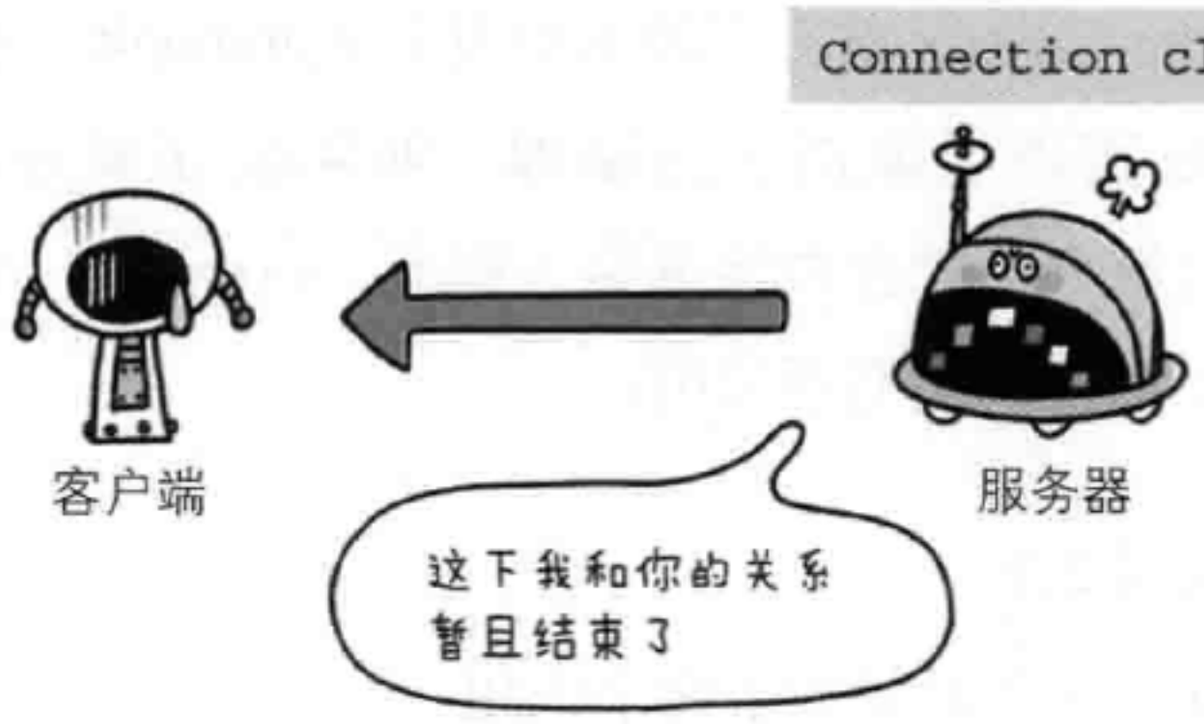
■控制不再转发给代理的首部字段




```
Connection: 不再转发的首部字段名
```

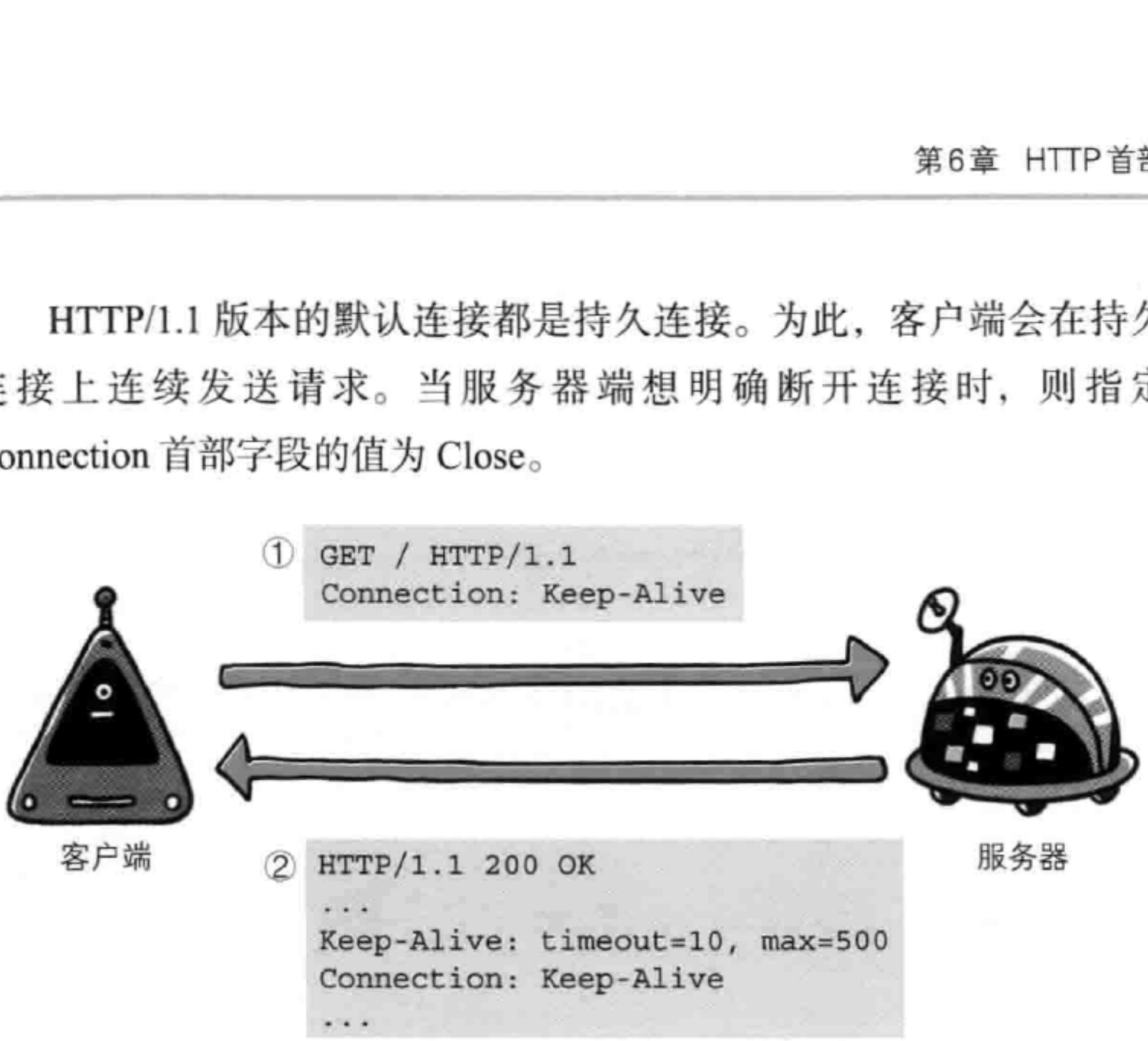

在客户端发送请求和服务器返回响应内，使用 Connection 首部字段，可控制不再转发给代理的首部字段（即 Hop-by-hop 首部）。

■管理持久连接




```
Connection: close
```

HTTP/1.1 版本的默认连接都是持久连接。为此，客户端会在持久连接上连续发送请求。当服务器端想明确断开连接时，则指定 Connection 首部字段的值为 Close。




```
Connection: Keep-Alive
```


HTTP/1.1 之前的 HTTP 版本的默认连接都是非持久连接。为此，如果想在旧版本的 HTTP 协议上维持持续连接，则需要指定 Connection 首部字段的值为 Keep-Alive。

如上图①所示，客户端发送请求给服务器时，服务器端会像上图②那样加上首部字段 Keep-Alive 及首部字段 Connection 后返回响应。

6.3.3 Date

首部字段 Date 表明创建 HTTP 报文的日期和时间。



HTTP/1.1 协议使用在 RFC1123 中规定的日期时间的格式，如下示例。

```
Date: Tue, 03 Jul 2012 04:40:59 GMT
```

之前的 HTTP 协议版本中使用在 RFC850 中定义的格式，如下所示。

```
Date: Tue, 03-Jul-12 04:40:59 GMT
```

除此之外，还有一种格式。它与 C 标准库内的 asctime() 函数的输出格式一致。


```
Date: Tue Jul 03 04:40:59 2012
```

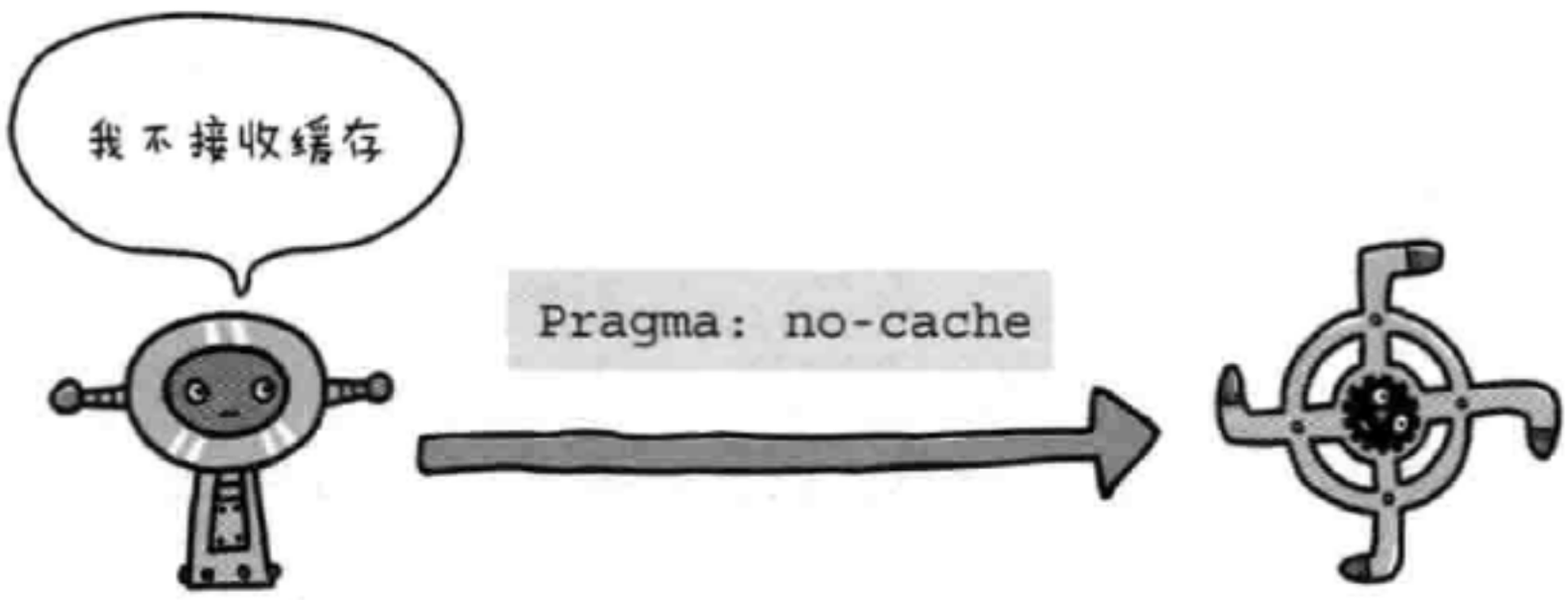
6.3.4 Pragma

Pragma 是 HTTP/1.1 之前版本的历史遗留字段，仅作为与 HTTP/1.0 的向后兼容而定义。

规范定义的形式唯一，如下所示。

```
Pragma: no-cache
```

该首部字段属于通用首部字段，但只用在客户端发送的请求中。客户端会要求所有的中间服务器不返回缓存的资源。



所有的中间服务器如果都能以HTTP/1.1为基准，那直接采用Cache-Control: no-cache指定缓存的处理方式是最为理想的。但要整体掌握全部中间服务器使用的HTTP协议版本却是不现实的。因此，发送的请求会同时含有下面两个首部字段。

```
Cache-Control: no-cache
Pragma: no-cache
```


6.3.5 Trailer



首部字段Trailer会事先说明在报文主体后记录了哪些首部字段。该首部字段可应用在HTTP/1.1版本分块传输编码时。

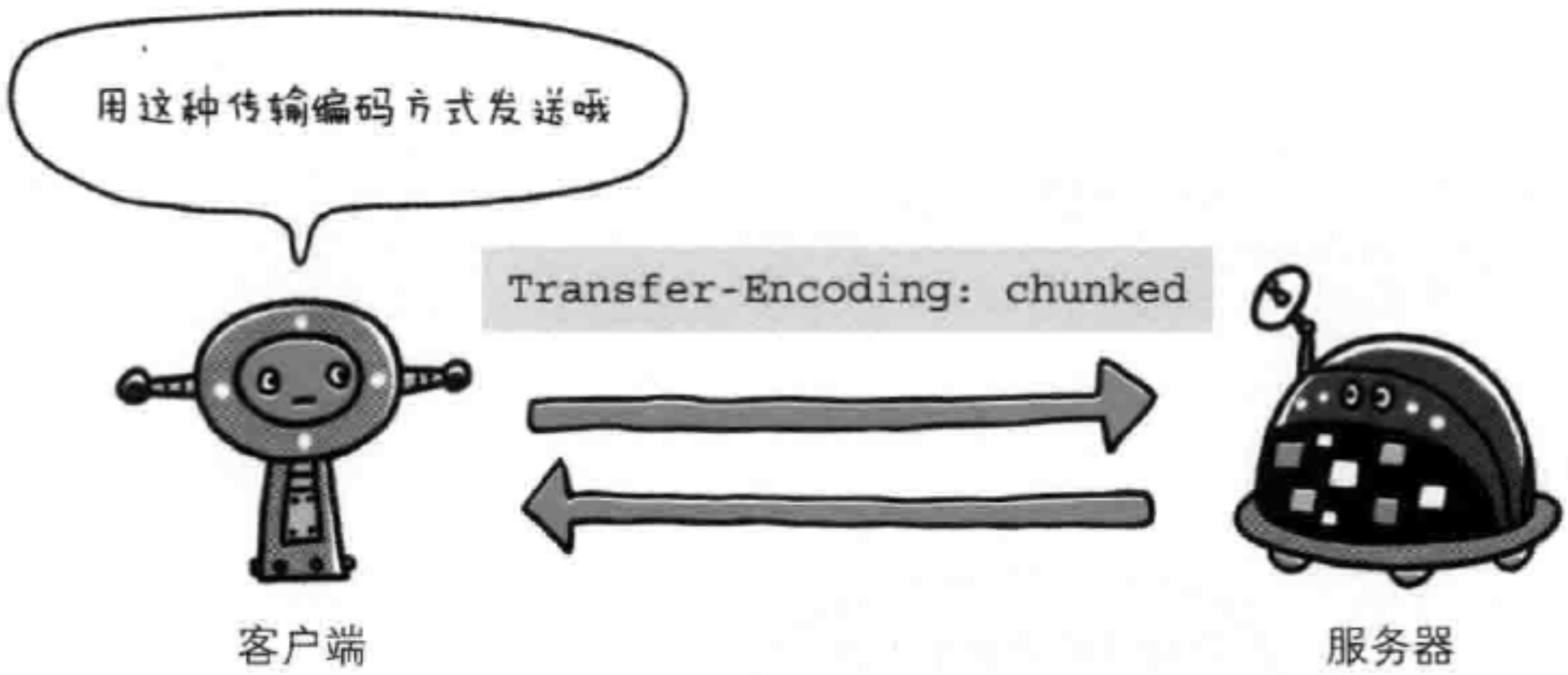
```
HTTP/1.1 200 OK
Date: Tue, 03 Jul 2012 04:40:56 GMT
Content-Type: text/html
```

```
...
Transfer-Encoding: chunked
Trailer: Expires

...(报文主体)...
0
Expires: Tue, 28 Sep 2004 23:59:59 GMT
```

以上用例中，指定首部字段 Trailer 的值为 Expires，在报文主体之后（分块长度 0 之后）出现了首部字段 Expires。

6.3.6 Transfer-Encoding



首部字段 Transfer-Encoding 规定了传输报文主体时采用的编码方式。

HTTP/1.1 的传输编码方式仅对分块传输编码有效。

```
HTTP/1.1 200 OK
Date: Tue, 03 Jul 2012 04:40:56 GMT
Cache-Control: public, max-age=604800
Content-Type: text/javascript; charset=utf-8
Expires: Tue, 10 Jul 2012 04:40:56 GMT
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
Content-Encoding: gzip
```

```
Transfer-Encoding: chunked
Connection: keep-alive

cf0    ←16进制(10进制为3312)

···3312字节分块数据···

392    ←16进制(10进制为914)

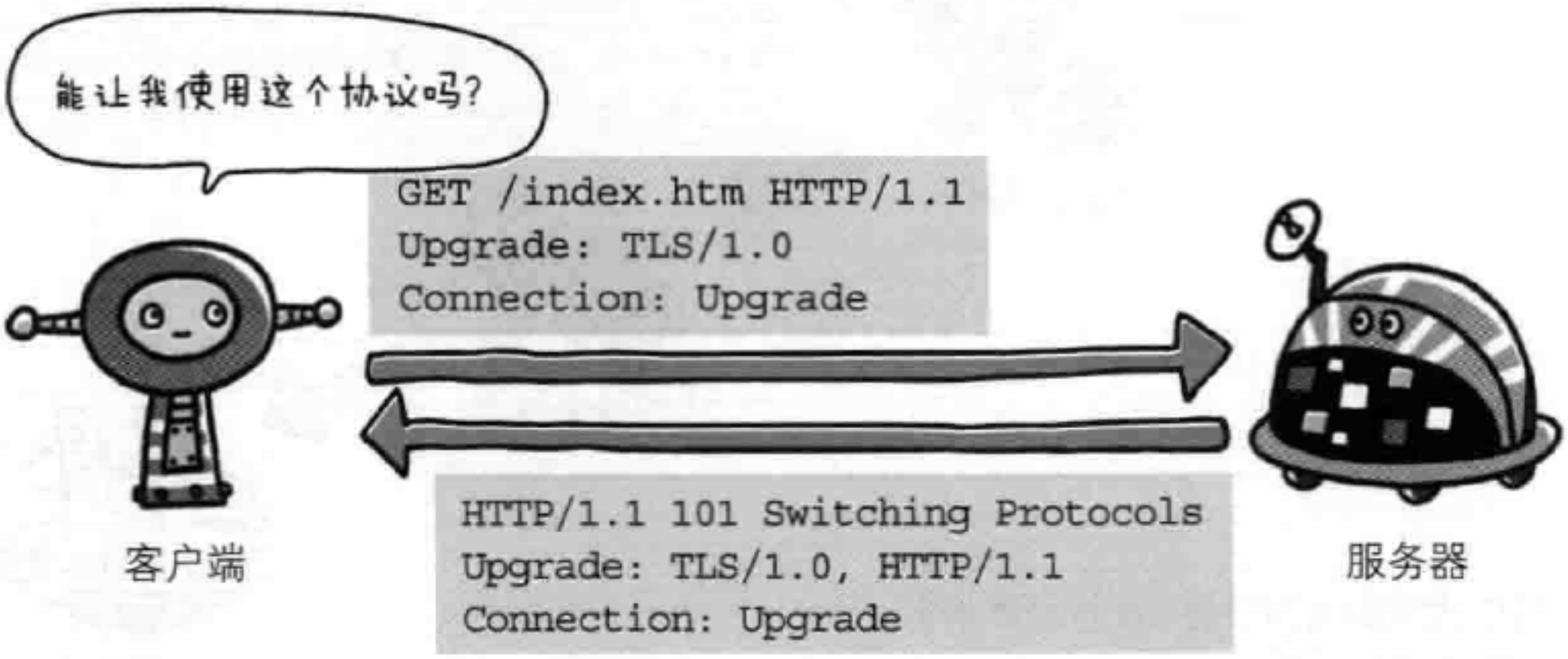
···914字节分块数据···

0
```

以上用例中，正如在首部字段 Transfer-Encoding 中指定的那样，有效使用分块传输编码，且分别被分成 3312 字节和 914 字节大小的分块数据。

6.3.7 Upgrade

首部字段 Upgrade 用于检测 HTTP 协议及其他协议是否可使用更高的版本进行通信，其参数值可以用来指定一个完全不同的通信协议。



上图用例中，首部字段 Upgrade 指定的值为 TLS/1.0。请注意此处两个字段首部字段的对应关系，Connection 的值被指定为 Upgrade。Upgrade 首部字段产生作用的 Upgrade 对象仅限于客户端和邻接服务器

之间。因此，使用首部字段 Upgrade 时，还需要额外指定 Connection: Upgrade。

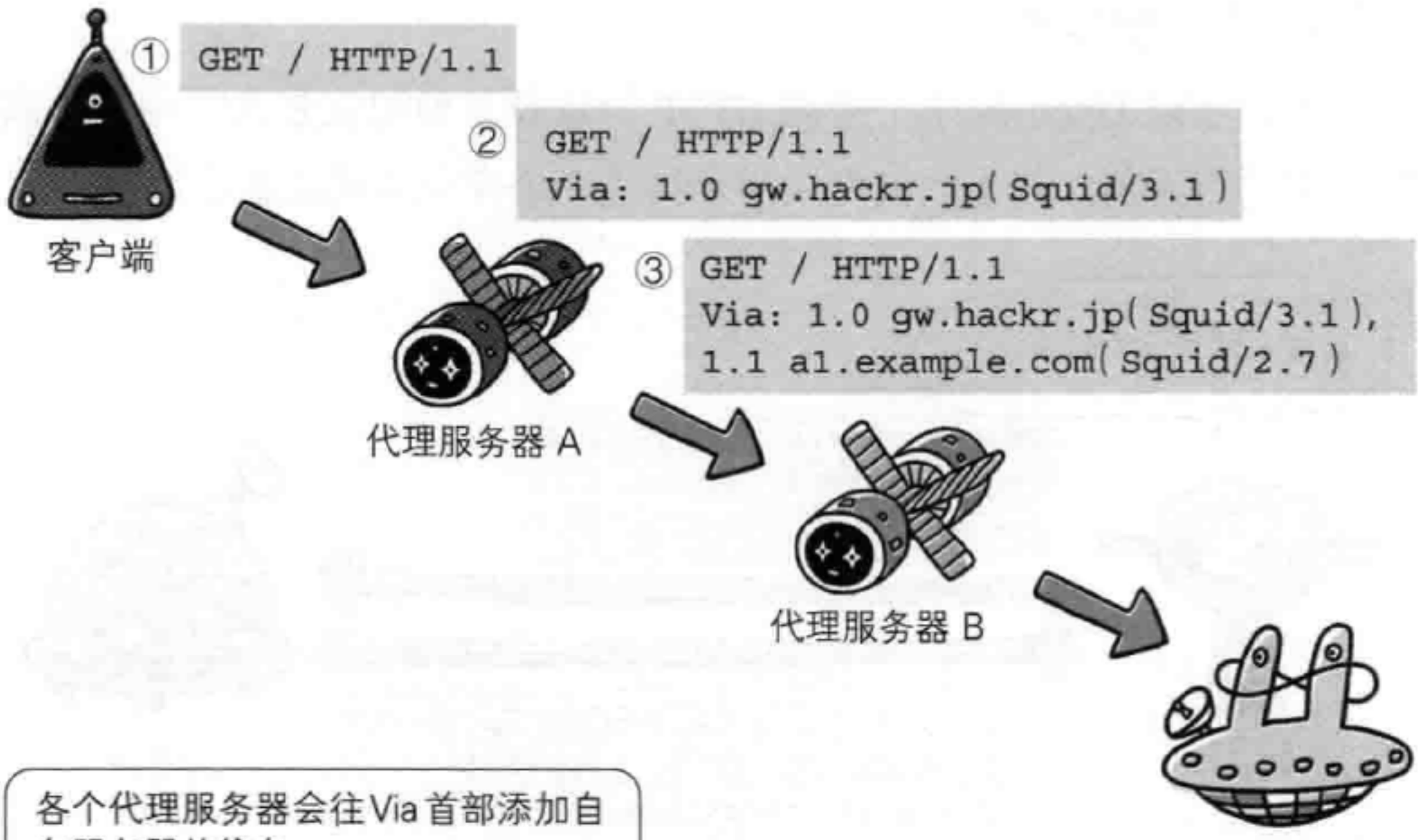
对于附有首部字段 Upgrade 的请求，服务器可用 101 Switching Protocols 状态码作为响应返回。

6.3.8 Via

使用首部字段 Via 是为了追踪客户端与服务器之间的请求和响应报文的传输路径。

报文经过代理或网关时，会先在首部字段 Via 中附加该服务器的信息，然后再进行转发。这个做法和 traceroute 及电子邮件的 Received 首部的工作机制很类似。

首部字段 Via 不仅用于追踪报文的转发，还可避免请求回环的发生。所以必须在经过代理时附加该首部字段内容。



上图用例中，在经过代理服务器 A 时，Via 首部附加了“1.0 gw.hackr.jp (Squid/3.1)”这样的字符串值。行头的 1.0 是指接收请求的服务器上应用的 HTTP 协议版本。接下来经过代理服务器 B 时亦是如此，

在 Via 首部附加服务器信息，也可增加 1 个新的 Via 首部写入服务器信息。

Via 首部是为了追踪传输路径，所以经常会和 TRACE 方法一起使用。比如，代理服务器接收到由 TRACE 方法发送过来的请求（其中 Max-Forwards: 0）时，代理服务器就不能再转发该请求了。这种情况下，代理服务器会将自身的信息附加到 Via 首部后，返回该请求的响应。

6.3.9 Warning

HTTP/1.1 的 Warning 首部是从 HTTP/1.0 的响应首部（Retry-After）演变过来的。该首部通常会告知用户一些与缓存相关的问题的警告。

```
Warning: 113 gw.hackr.jp:8080 "Heuristic expiration" Tue, 03 Jul ⇒
2012 05:09:44 GMT
```


Warning 首部的格式如下。最后的日期时间部分可省略。

```
Warning: [警告码][警告的主机：端口号] "[警告内容]" ([日期时间])
```

HTTP/1.1 中定义了 7 种警告。警告码对应的警告内容仅推荐参考。另外，警告码具备扩展性，今后有可能追加新的警告码。

表 6-7：HTTP/1.1 警告码

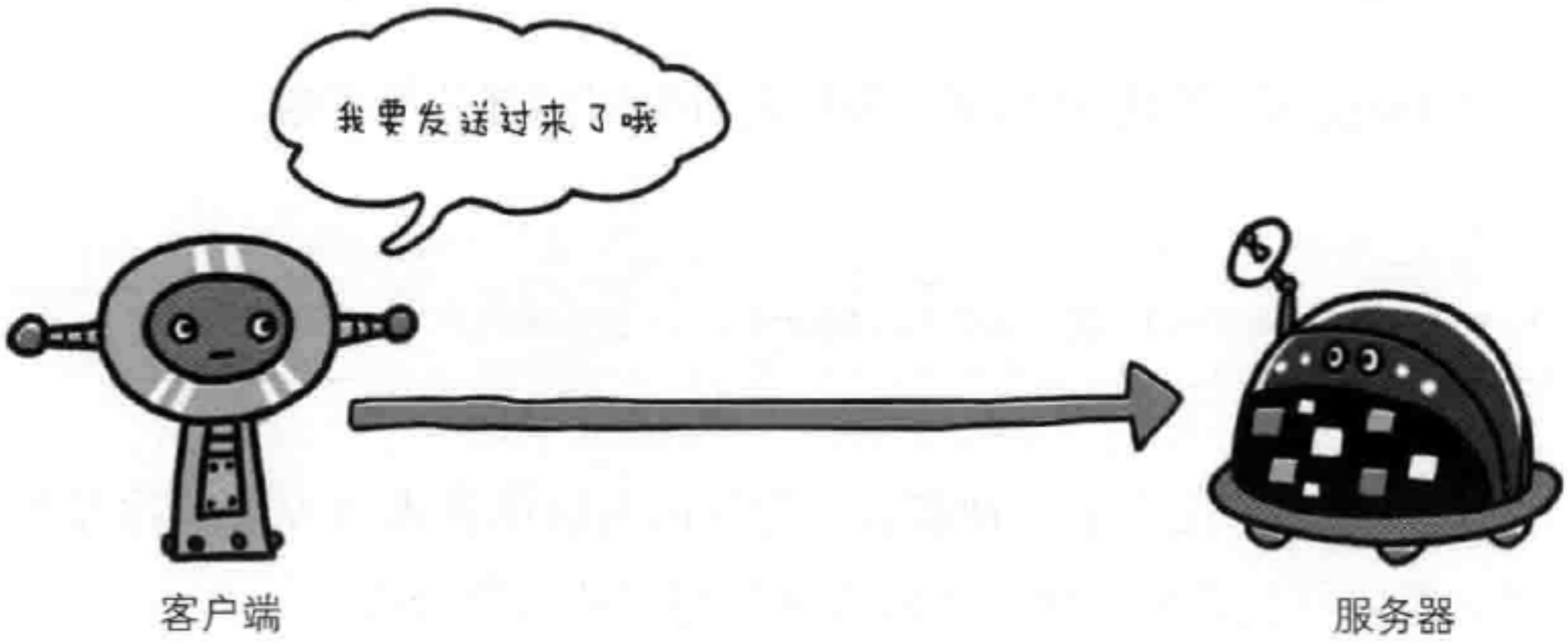
警告码	警告内容	说明
110	Response is stale（响应已过期）	代理返回已过期的资源
111	Revalidation failed（再验证失败）	代理再验证资源有效性时失败（服务器无法到达等原因）
112	Disconnection operation（断开连接操作）	代理与互联网连接被故意切断

（续）

警告码	警告内容	说明
113	Heuristic expiration（试探性过期）	响应的使用期超过24小时（有效缓存的设定时间大于24小时的情况下）
199	Miscellaneous warning（杂项警告）	任意的警告内容
214	Transformation applied（使用了转换）	代理对内容编码或媒体类型等执行了某些处理时
299	Miscellaneous persistent warning（持久杂项警告）	任意的警告内容

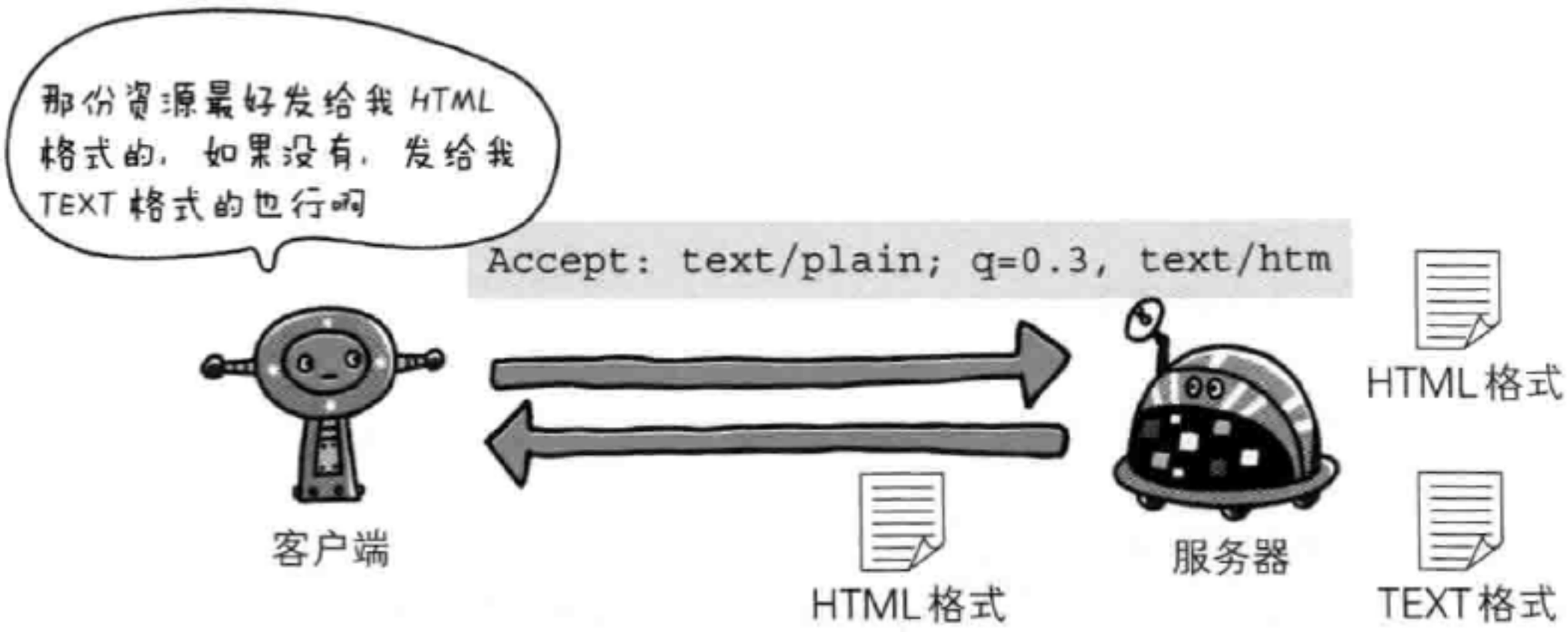
6.4 请求首部字段

请求首部字段是从客户端往服务器端发送请求报文中所使用的字段，用于补充请求的附加信息、客户端信息、对响应内容相关的优先级等内容。



图：HTTP 请求报文中使用的首部字段

6.4.1 Accept




```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Accept 首部字段可通知服务器，用户代理能够处理的媒体类型及媒体类型的相对优先级。可使用 type/subtype 这种形式，一次指定多种媒体类型。

下面我们试举几个媒体类型的例子。

- 文本文件

  text/html, text/plain, text/css ...

  application/xhtml+xml, application/xml ...

- 图片文件

  image/jpeg, image/gif, image/png ...

- 视频文件

  video/mpeg, video/quicktime ...

- 应用程序使用的二进制文件

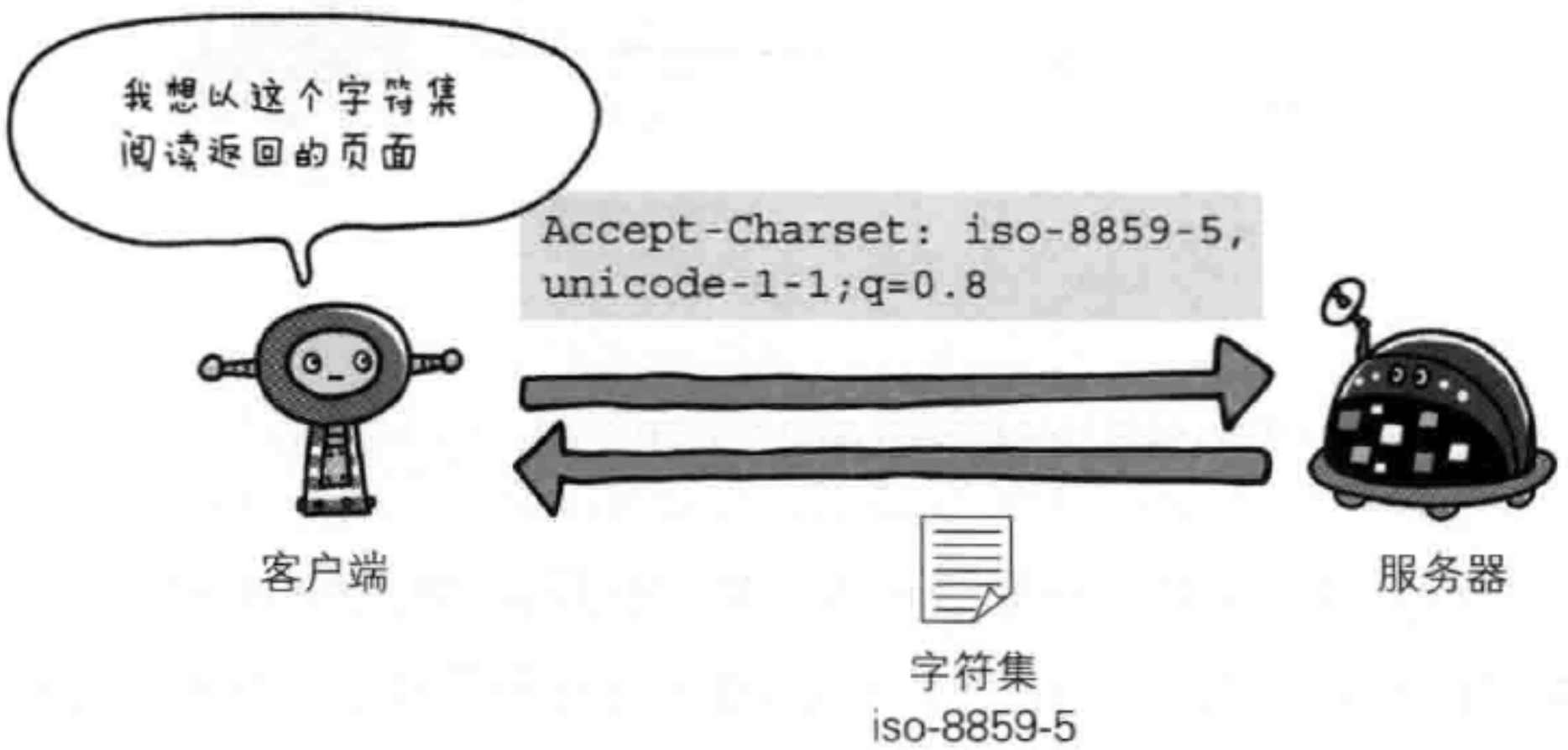
  application/octet-stream, application/zip ...

比如，如果浏览器不支持 PNG 图片的显示，那 Accept 就不指定 image/png，而指定可处理的 image/gif 和 image/jpeg 等图片类型。

若想要给显示的媒体类型增加优先级，则使用 q= 来额外表示权重值[①]，用分号（;）进行分隔。权重值 q 的范围是 0~1（可精确到小数点后 3 位），且 1 为最大值。不指定权重 q 值时，默认权重为 q=1.0。

当服务器提供多种内容时，将会首先返回权重值最高的媒体类型。

6.4.2 Accept-Charset




```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

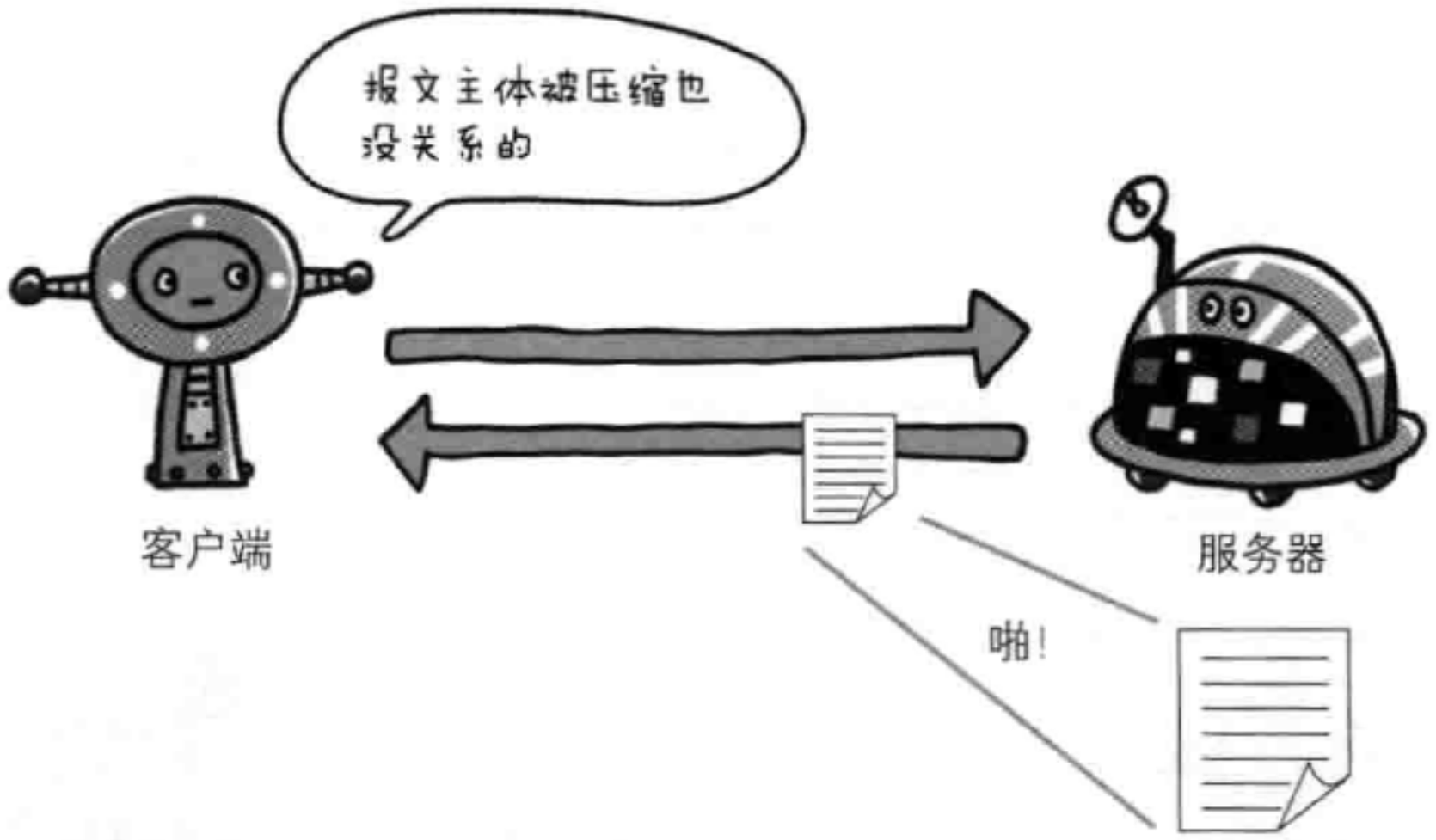
Accept-Charset 首部字段可用来通知服务器用户代理支持的字符集及字符集的相对优先顺序。另外，可一次性指定多种字符集。与首部字段 Accept 相同的是可用权重 q 值来表示相对优先级。

该首部字段应用于内容协商机制的服务器驱动协商。

---

① 原文是“品質係数”。在RFC2616定义中，此处的q是指qvalue，即quality factor。直译的话就是质量数，但经过综合考虑理解记忆的便利性后，似乎采用权重值更为稳妥。——译者注

6.4.3 Accept-Encoding




```
Accept-Encoding: gzip, deflate
```


Accept-Encoding 首部字段用来告知服务器用户代理支持的内容编码及内容编码的优先级顺序。可一次性指定多种内容编码。

下面试举出几个内容编码的例子。

- gzip

  由文件压缩程序 gzip（GNU zip）生成的编码格式（RFC1952），采用 Lempel-Ziv 算法（LZ77）及 32 位循环冗余校验（Cyclic Redundancy Check，通称 CRC）。

- compress

  由 UNIX 文件压缩程序 compress 生成的编码格式，采用 Lempel-Ziv-Welch 算法（LZW）。

- deflate

  组合使用 zlib 格式（RFC1950）及由 deflate 压缩算法（RFC1951）生成的编码格式。

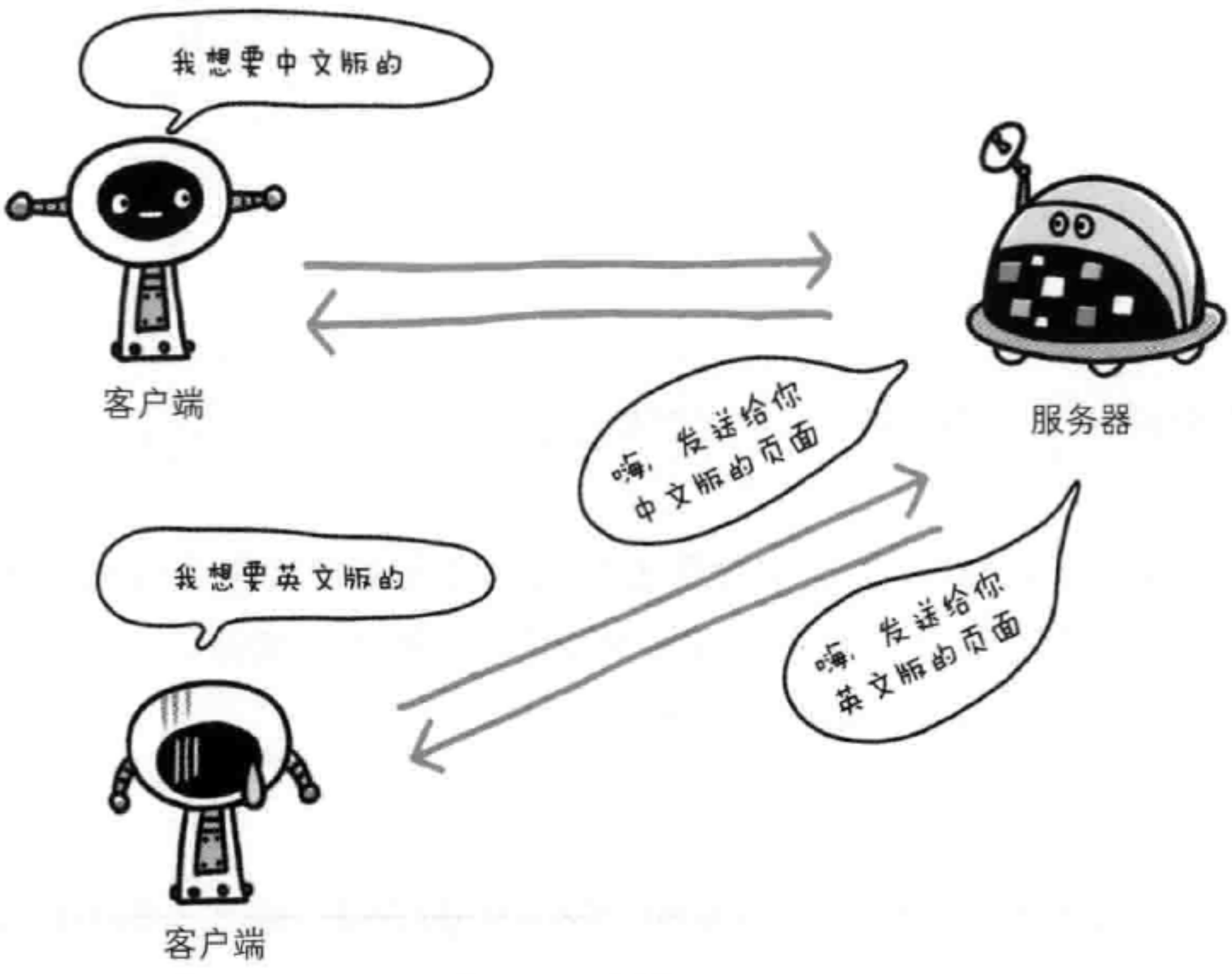
- identity

  不执行压缩或不会变化的默认编码格式

采用权重 q 值来表示相对优先级，这点与首部字段 Accept 相同。

另外，也可使用星号（*）作为通配符，指定任意的编码格式。

6.4.4 Accept-Language



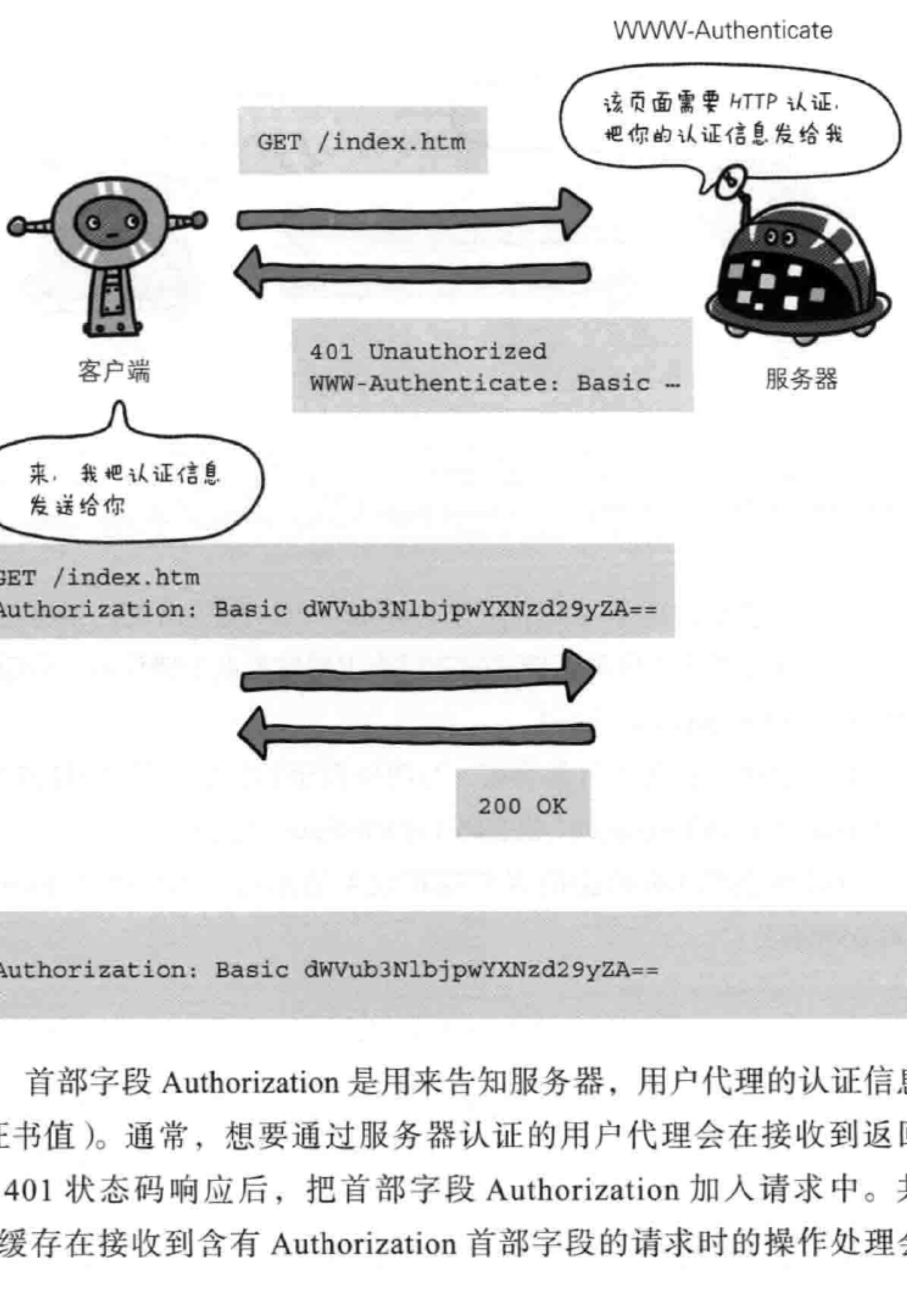

```
Accept-Language: zh-cn,zh;q=0.7,en-us,en;q=0.3
```

首部字段 Accept-Language 用来告知服务器用户代理能够处理的自然语言集（指中文或英文等），以及自然语言集的相对优先级。可一次指定多种自然语言集。

和 Accept 首部字段一样，按权重值 q 来表示相对优先级。在上述

图例中，客户端在服务器有中文版资源的情况下，会请求其返回中文版对应的响应，没有中文版时，则请求返回英文版响应。

6.4.5 Authorization



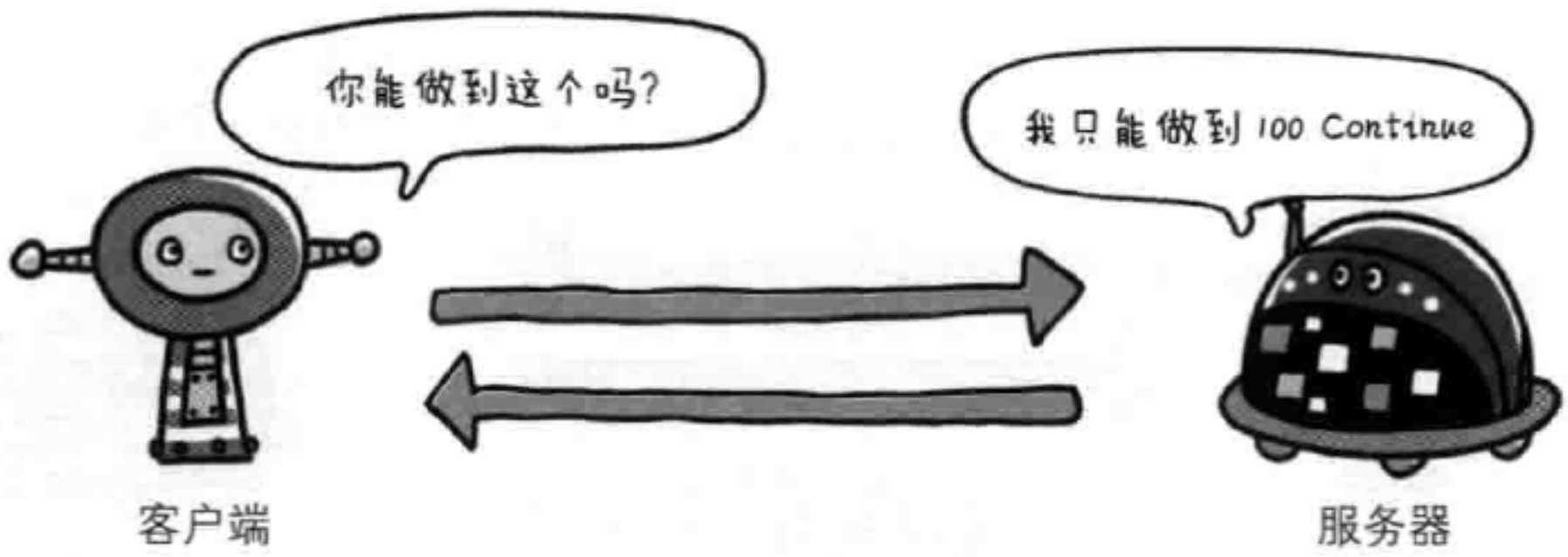

```
Authorization: Basic dWVub3NlbjpwYXNzd29yZA==
```

首部字段 Authorization 是用来告知服务器，用户代理的认证信息（证书值）。通常，想要通过服务器认证的用户代理会在接收到返回的 401 状态码响应后，把首部字段 Authorization 加入请求中。共用缓存在接收到含有 Authorization 首部字段的请求时的操作处理会

略有差异。

有关 HTTP 访问认证及 Authorization 首部字段，稍后的章节还会详细说明。另外，读者也可参阅 RFC2616。

6.4.6 Expect



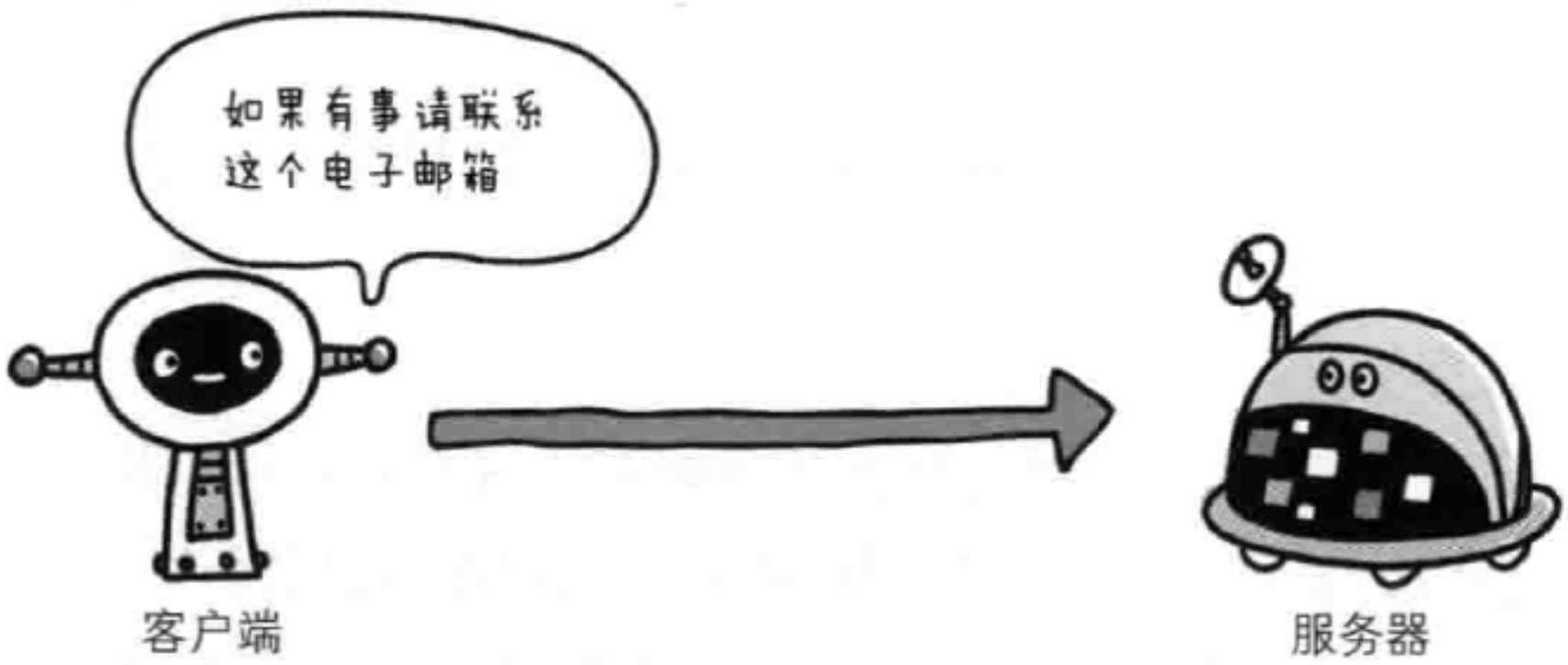

```
Expect: 100-continue
```

客户端使用首部字段 Expect 来告知服务器，期望出现的某种特定行为。因服务器无法理解客户端的期望作出回应而发生错误时，会返回状态码 417 Expectation Failed。

客户端可以利用该首部字段，写明所期望的扩展。虽然 HTTP/1.1 规范只定义了 100-continue（状态码 100 Continue 之意）。

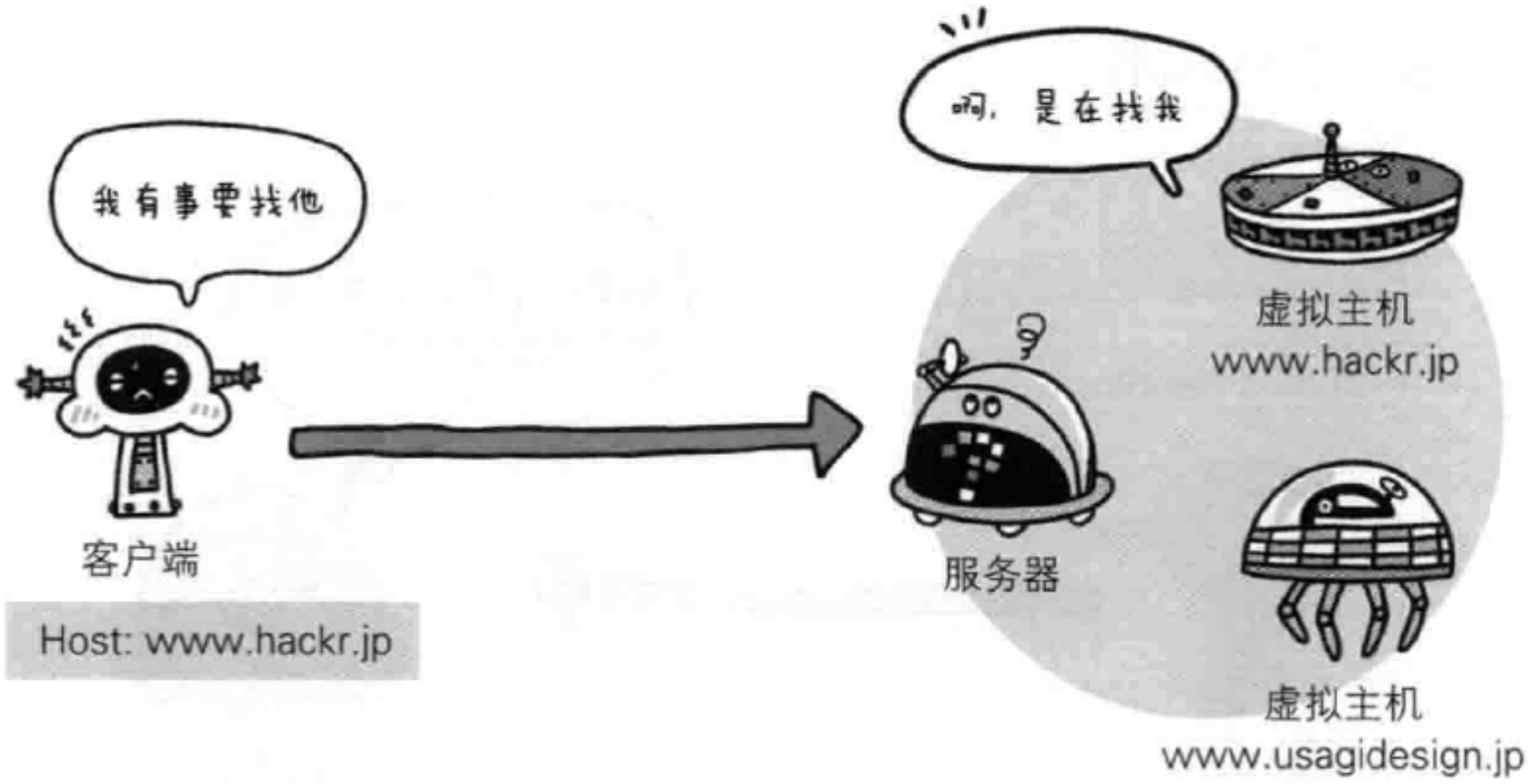
等待状态码 100 响应的客户端在发生请求时，需要指定 Expect: 100-continue。

6.4.7 From



首部字段 From 用来告知服务器使用用户代理的用户的电子邮件地址。通常，其使用目的就是为了显示搜索引擎等用户代理的负责人的电子邮件联系方式。使用代理时，应尽可能包含 From 首部字段（但可能会因代理不同，将电子邮件地址记录在 User-Agent 首部字段内）。

6.4.8 Host



图：虚拟主机运行在同一个 IP 上，因此使用首部字段 Host 加以区分

```
Host: www.hackr.jp
```

首部字段 Host 会告知服务器，请求的资源所处的互联网主机名和端口号。Host 首部字段在 HTTP/1.1 规范内是唯一一个必须被包含在请求内的首部字段。

首部字段 Host 和以单台服务器分配多个域名的虚拟主机的工作机制有很密切的关联，这是首部字段 Host 必须存在的意义。

请求被发送至服务器时，请求中的主机名会用 IP 地址直接替换解决。但如果这时，相同的 IP 地址下部署运行着多个域名，那么服务器就会无法理解究竟是哪个域名对应的请求。因此，就需要使用首部字段 Host 来明确指出请求的主机名。若服务器未设定主机名，那直接发送一个空值即可。如下所示。


```
Host:
```

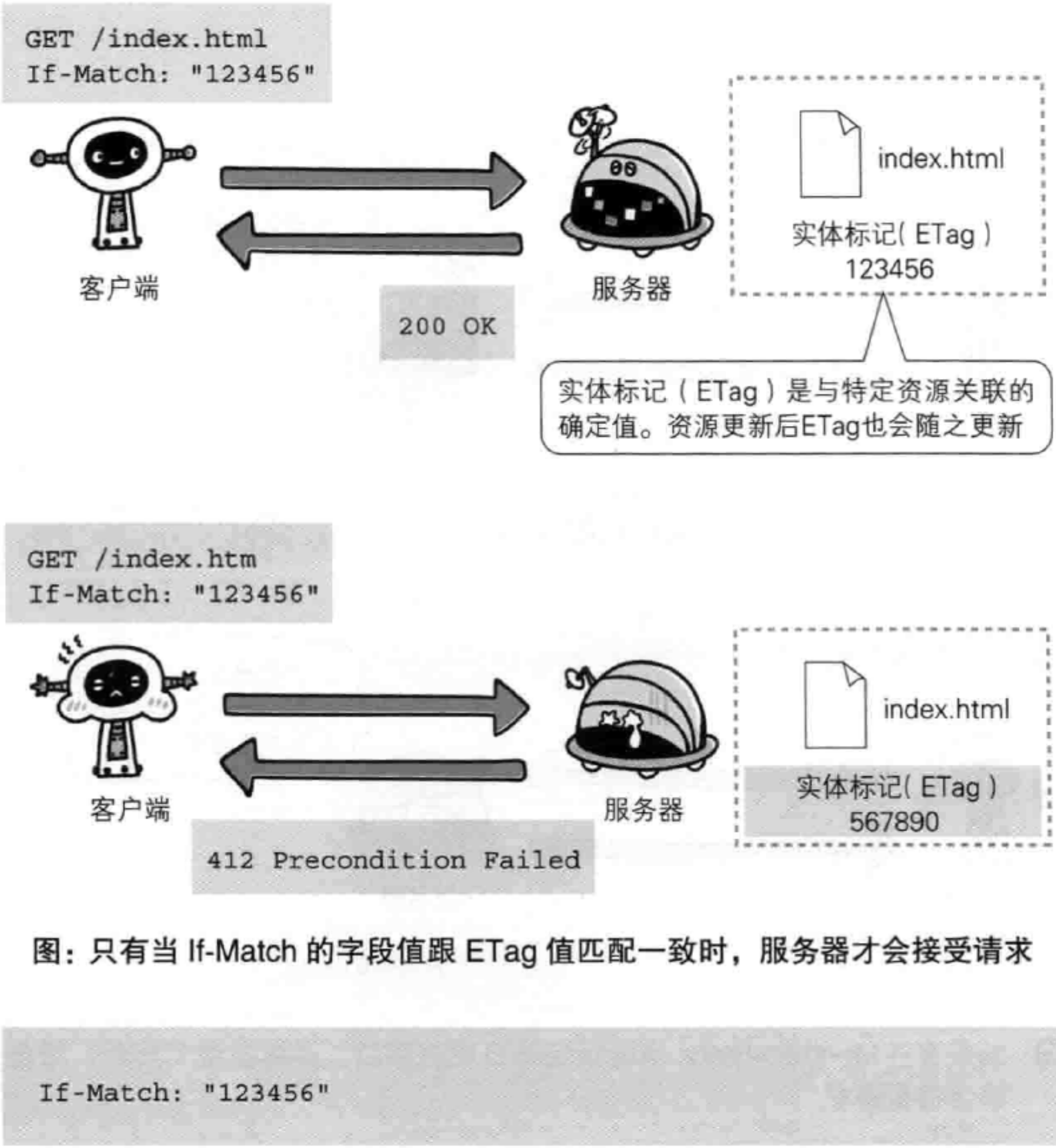
6.4.9 If-Match

If-Match
If-Modified-Since
If-None-Match
If-Range
If-Unmodified-Since



图：附带条件请求

形如 If-xxx 这种样式的请求首部字段，都可称为条件请求。服务器接收到附带条件的请求后，只有判断指定条件为真时，才会执行请求。



图：只有当 If-Match 的字段值跟 ETag 值匹配一致时，服务器才会接受请求

```
If-Match: "123456"
```

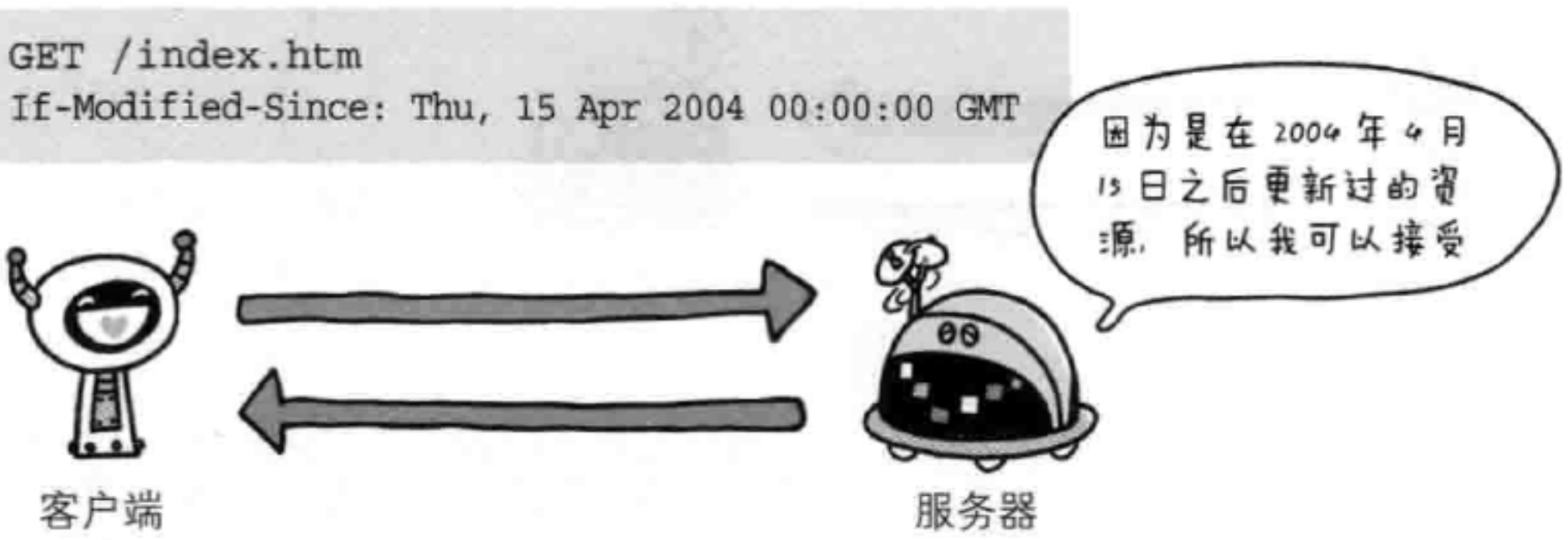
首部字段 If-Match，属附带条件之一，它会告知服务器匹配资源所用的实体标记（ETag）值。这时的服务器无法使用弱 ETag 值。（请参照本章有关首部字段 ETag 的说明）

服务器会比对 If-Match 的字段值和资源的 ETag 值，仅当两者一致

时，才会执行请求。反之，则返回状态码 412 Precondition Failed 的响应。

还可以使用星号（*）指定 If-Match 的字段值。针对这种情况，服务器将会忽略 ETag 的值，只要资源存在就处理请求。

6.4.10 If-Modified-Since



图：如果在 If-Modified-Since 字段指定的日期时间后，资源发生了更新，服务器会接受请求

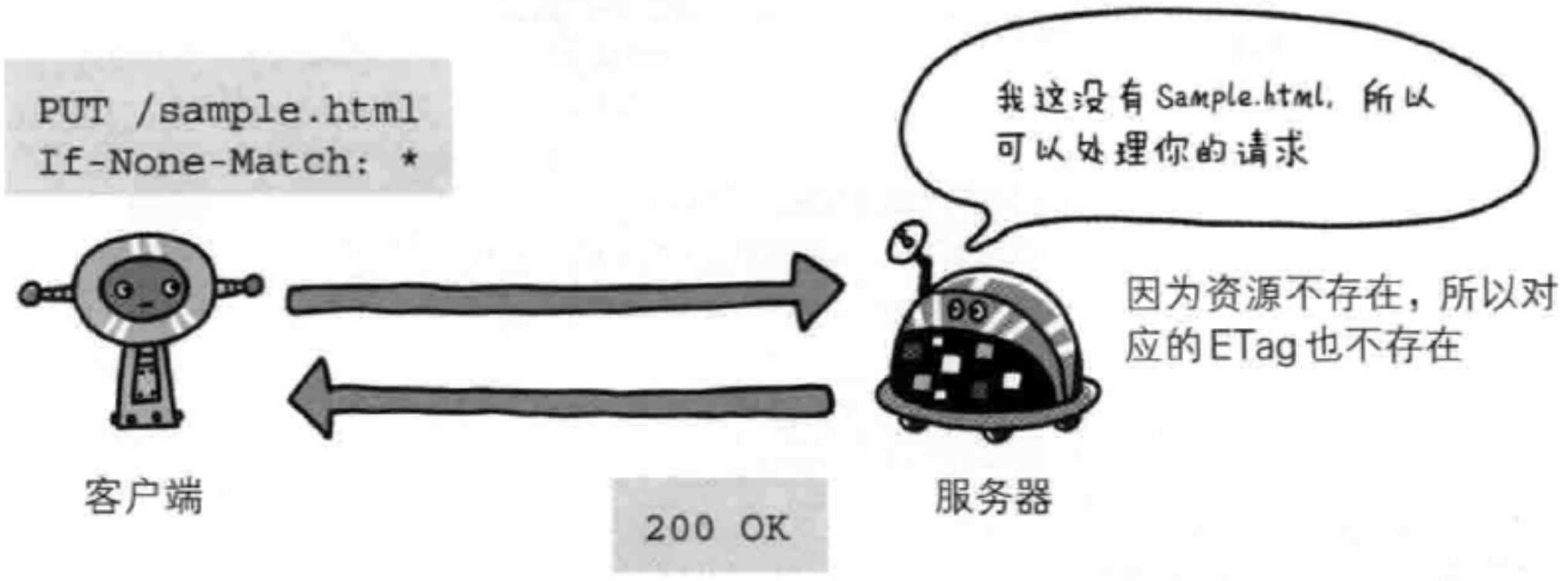
```
If-Modified-Since: Thu, 15 Apr 2004 00:00:00 GMT
```

首部字段 If-Modified-Since，属附带条件之一，它会告知服务器若 If-Modified-Since 字段值早于资源的更新时间，则希望能处理该请求。

而在指定 If-Modified-Since 字段值的日期时间之后，如果请求的资源都没有过更新，则返回状态码 304 Not Modified 的响应。

If-Modified-Since 用于确认代理或客户端拥有的本地资源的有效性。获取资源的更新日期时间，可通过确认首部字段 Last-Modified 来确定。

6.4.11 If-None-Match



图：只有在 If-None-Match 的字段值与 ETag 值不一致时，可处理该请求。与 If-Match 首部字段的作用相反

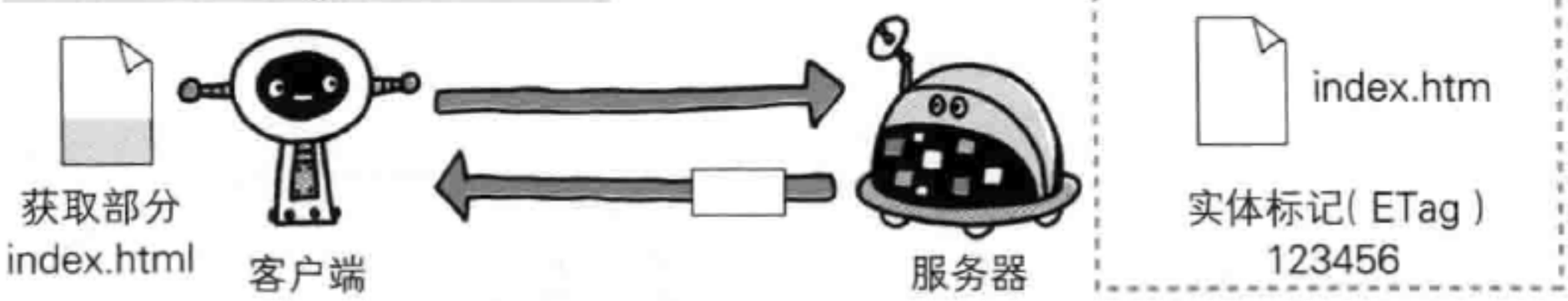
首部字段 If-None-Match 属于附带条件之一。它和首部字段 If-Match 作用相反。用于指定 If-None-Match 字段值的实体标记（ETag）值与请求资源的 ETag 不一致时，它就告知服务器处理该请求。

在 GET 或 HEAD 方法中使用首部字段 If-None-Match 可获取最新的资源。因此，这与使用首部字段 If-Modified-Since 时有些类似。

6.4.12 If-Range

If-Range字段值若是跟ETag值或更新的日期时间匹配一致，那么就作为范围请求处理

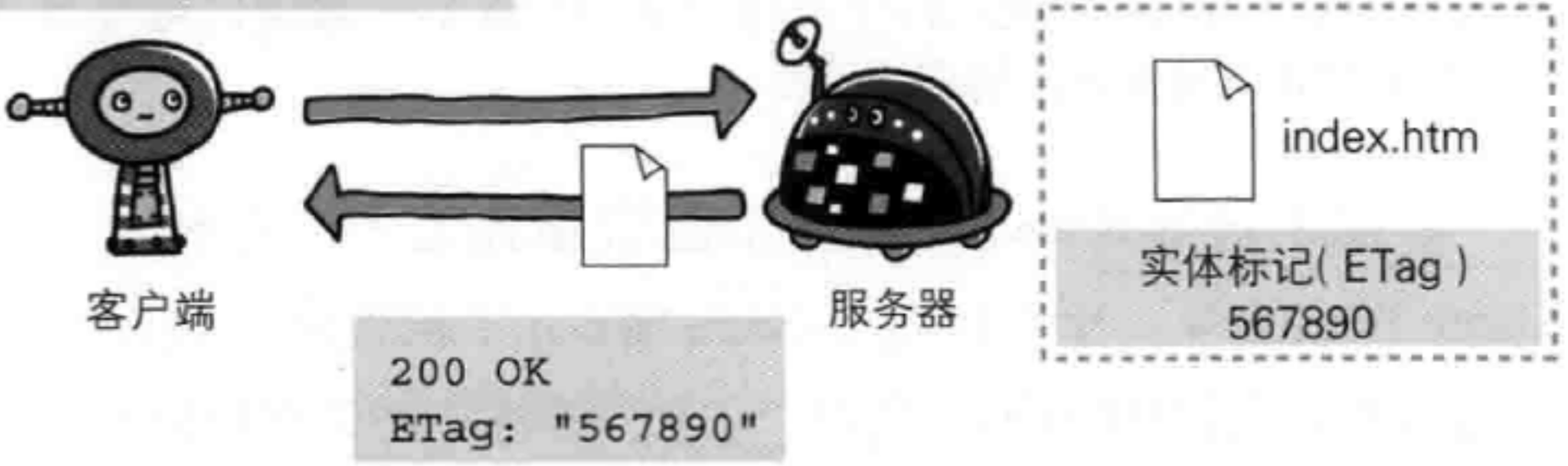
```
GET /index.html
If-Range: "123456"
Range: bytes=5001-10000
```




```
206 Partial Content
Content-Range: bytes 5001-10000/10000
Content-Length: 5000
```

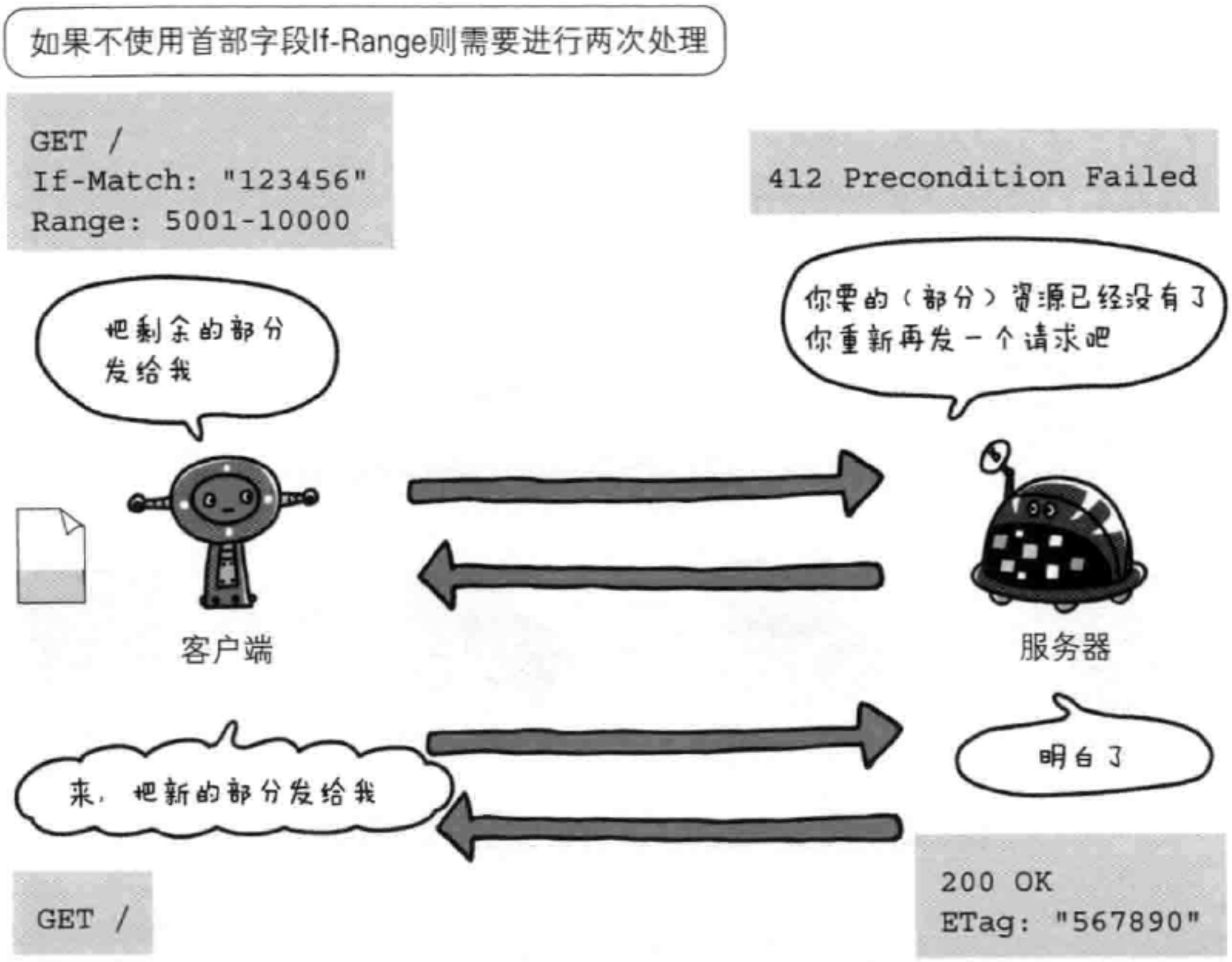
若不一致，则忽略范围请求，返回全部资源

```
GET /index.html
If-Range: "123456"
Range: bytes=5001-10000
```




```
200 OK
ETag: "567890"
```

首部字段 If-Range 属于附带条件之一。它告知服务器若指定的 If-Range 字段值（ETag 值或者时间）和请求资源的 ETag 值或时间相一致时，则作为范围请求处理。反之，则返回全体资源。



下面我们思考一下不使用首部字段 If-Range 发送请求的情况。服务器端的资源如果更新，那客户端持有资源中的一部分也会随之无效，当然，范围请求作为前提是无效的。这时，服务器会暂且以状态码 412 Precondition Failed 作为响应返回，其目的是催促客户端再次发送请求。这样一来，与使用首部字段 If-Range 比起来，就需要花费两倍的功夫。

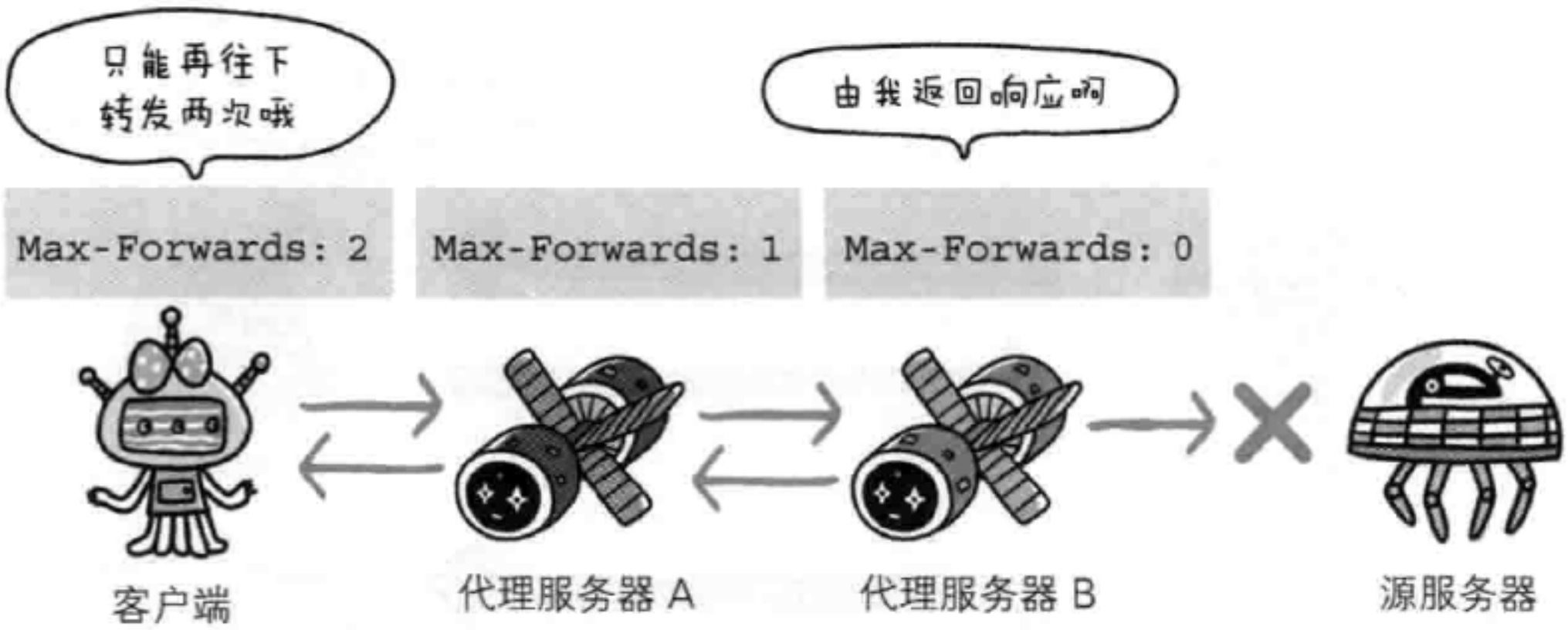
6.4.13 If-Unmodified-Since

```
If-Unmodified-Since: Thu, 03 Jul 2012 00:00:00 GMT
```

首部字段 If-Unmodified-Since 和首部字段 If-Modified-Since 的作用相反。它的作用的是告知服务器，指定的请求资源只有在字段值内指定的日期时间之后，未发生更新的情况下，才能处理请求。如果在指

定日期时间后发生了更新，则以状态码 412 Precondition Failed 作为响应返回。

6.4.14 Max-Forwards



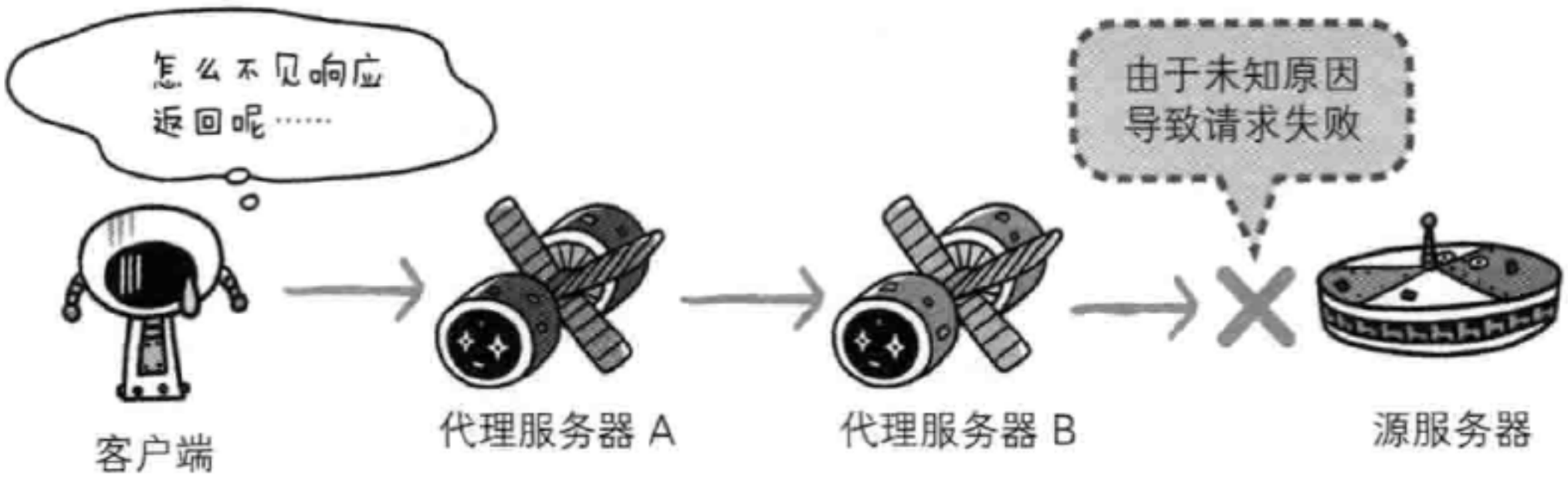
图：每次转发数值减 1。当数值变 0 时返回响应


```
Max-Forwards: 10
```

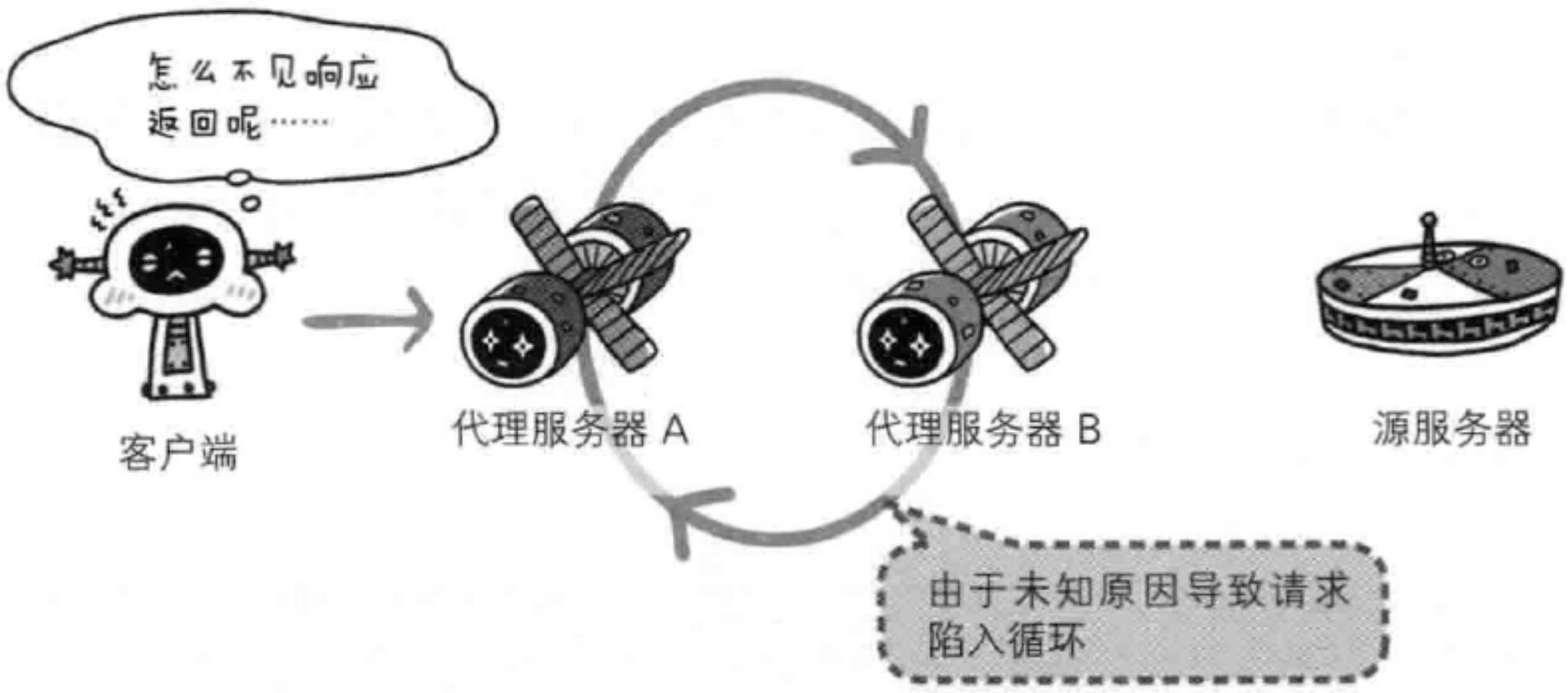
通过 TRACE 方法或 OPTIONS 方法，发送包含首部字段 Max-Forwards 的请求时，该字段以十进制整数形式指定可经过的服务器最大数目。服务器在往下一个服务器转发请求之前，Max-Forwards 的值减 1 后重新赋值。当服务器接收到 Max-Forwards 值为 0 的请求时，则不再进行转发，而是直接返回响应。

使用 HTTP 协议通信时，请求可能会经过代理等多台服务器。途中，如果代理服务器由于某些原因导致请求转发失败，客户端也就等不到服务器返回的响应了。对此，我们无从可知。

可以灵活使用首部字段 Max-Forwards，针对以上问题产生的原因展开调查。由于当 Max-Forwards 字段值为 0 时，服务器就会立即返回响应，由此我们至少可以对以那台服务器为终点的传输路径的通信状况有所把握。



图：代理 B 到源服务器的请求失败了，但客户端不知道



图：由于未知原因，导致请求陷入代理之间的循环，但客户端不知道

6.4.15 Proxy-Authorization

```
Proxy-Authorization: Basic dGlwOjkpNLAGfFY5
```

接收到从代理服务器发来的认证质询时，客户端会发送包含首部字段 Proxy-Authorization 的请求，以告知服务器认证所需要的信息。

这个行为是与客户端和服务器之间的 HTTP 访问认证相类似的，不同之处在于，认证行为发生在客户端与代理之间。客户端与服务器之间的认证，使用首部字段 Authorization 可起到相同作用。有关 HTTP 访问

认证，后面的章节会作详尽阐述。

6.4.16 Range

```
Range: bytes=5001-10000
```

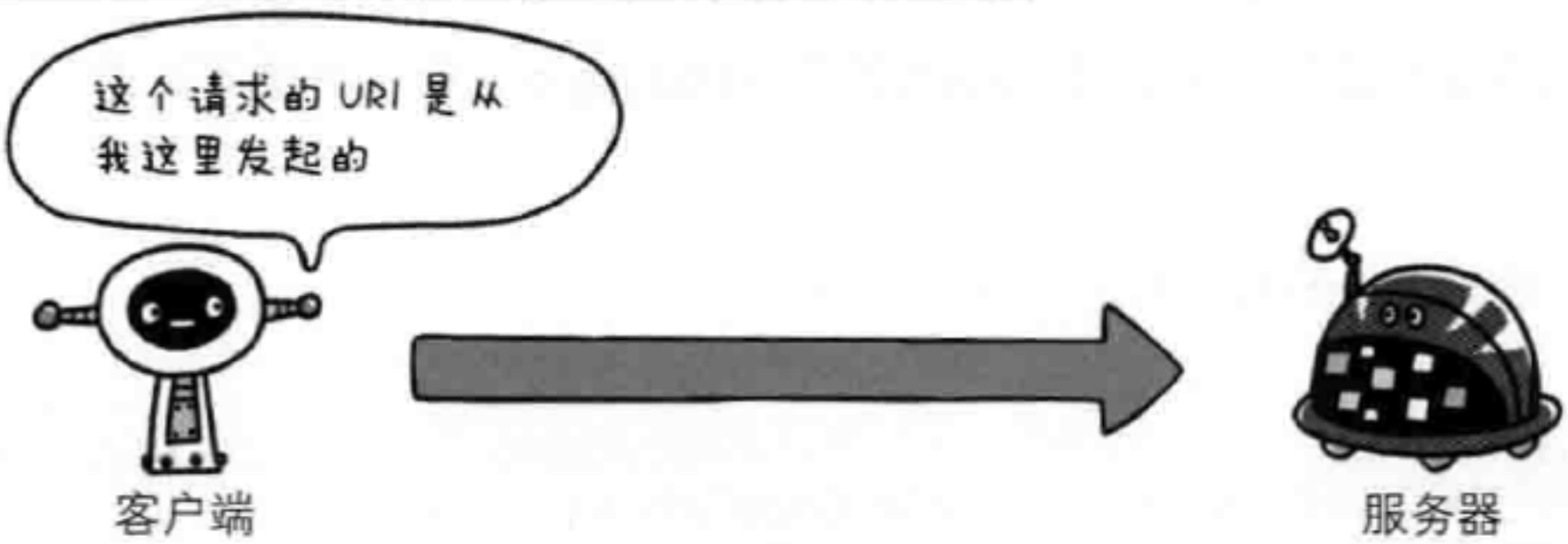
对于只需获取部分资源的范围请求，包含首部字段 Range 即可告知服务器资源的指定范围。上面的示例表示请求获取从第 5001 字节至第 10000 字节的资源。

接收到附带 Range 首部字段请求的服务器，会在处理请求之后返回状态码为 206 Partial Content 的响应。无法处理该范围请求时，则会返回状态码 200 OK 的响应及全部资源。

6.4.17 Referer

```
GET /
Referer: http://www.hackr.jp/index.htm
```

只要查看Referer就能知道请求的URI是从哪个Web页面发起的




```
Referer: http://www.hackr.jp/index.htm
```

首部字段 Referer 会告知服务器请求的原始资源的 URI。

客户端一般都会发送 Referer 首部字段给服务器。但当直接在浏览

器的地址栏输入 URI，或出于安全性的考虑时，也可以不发送该首部字段。

因为原始资源的 URI 中的查询字符串可能含有 ID 和密码等保密信息，要是写进 Referer 转发给其他服务器，则有可能导致保密信息的泄露。

另外，Referer 的正确的拼写应该是 Referrer，但不知为何，大家一直沿用这个错误的拼写。

6.4.18 TE

```
TE: gzip, deflate;q=0.5
```

首部字段 TE 会告知服务器客户端能够处理响应的传输编码方式及相对优先级。它和首部字段 Accept-Encoding 的功能很相像，但是用于传输编码。

首部字段 TE 除指定传输编码之外，还可以指定伴随 trailer 字段的分块传输编码的方式。应用后者时，只需把 trailers 赋值给该字段值。

```
TE: trailers
```

6.4.19 User-Agent



图：User-Agent 用于传达浏览器的种类

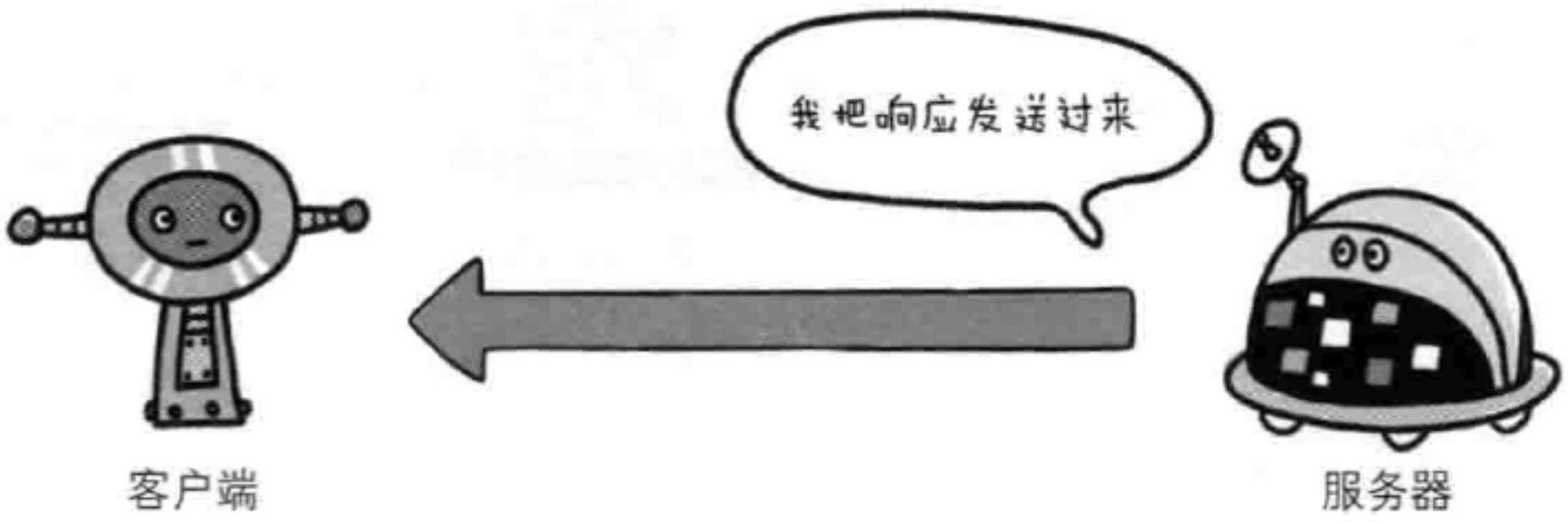
```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/⇒
20100101 Firefox/13.0.1
```

首部字段 User-Agent 会将创建请求的浏览器和用户代理名称等信息传达给服务器。

由网络爬虫发起请求时，有可能会在字段内添加爬虫作者的电子邮件地址。此外，如果请求经过代理，那么中间也很可能被添加上代理服务器的名称。

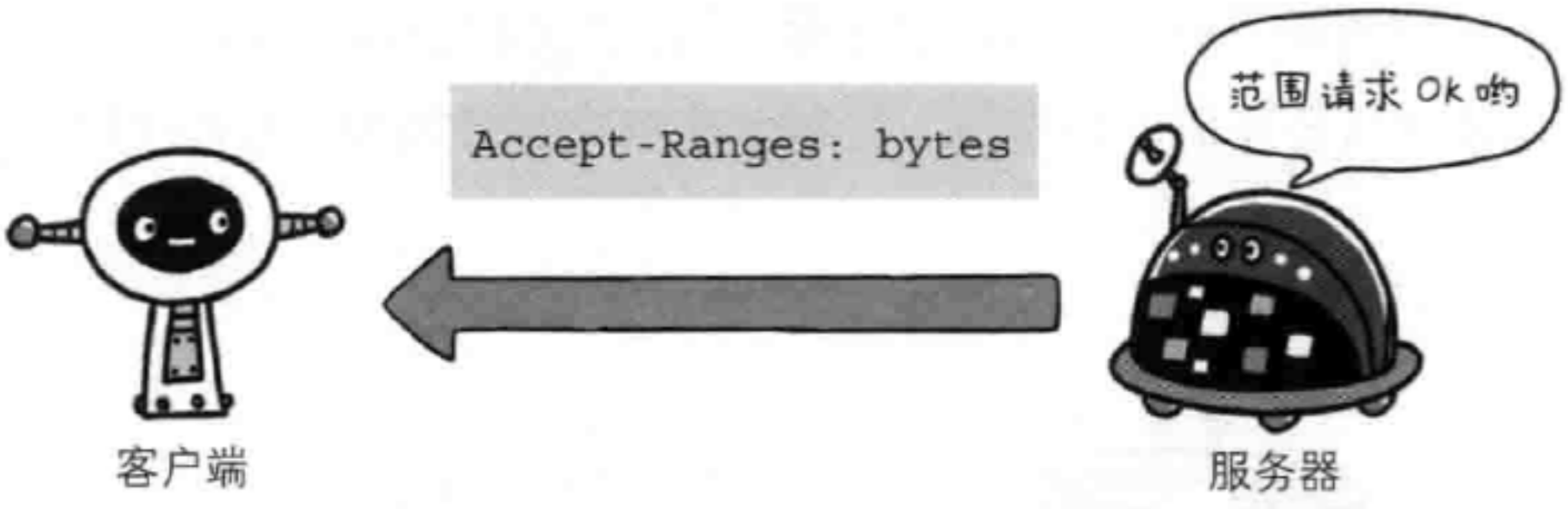
6.5 响应首部字段

响应首部字段是由服务器端向客户端返回响应报文中所使用的字段，用于补充响应的附加信息、服务器信息，以及对客户端的附加要求等信息。



图：HTTP 响应报文中使用的首部字段

6.5.1 Accept-Ranges



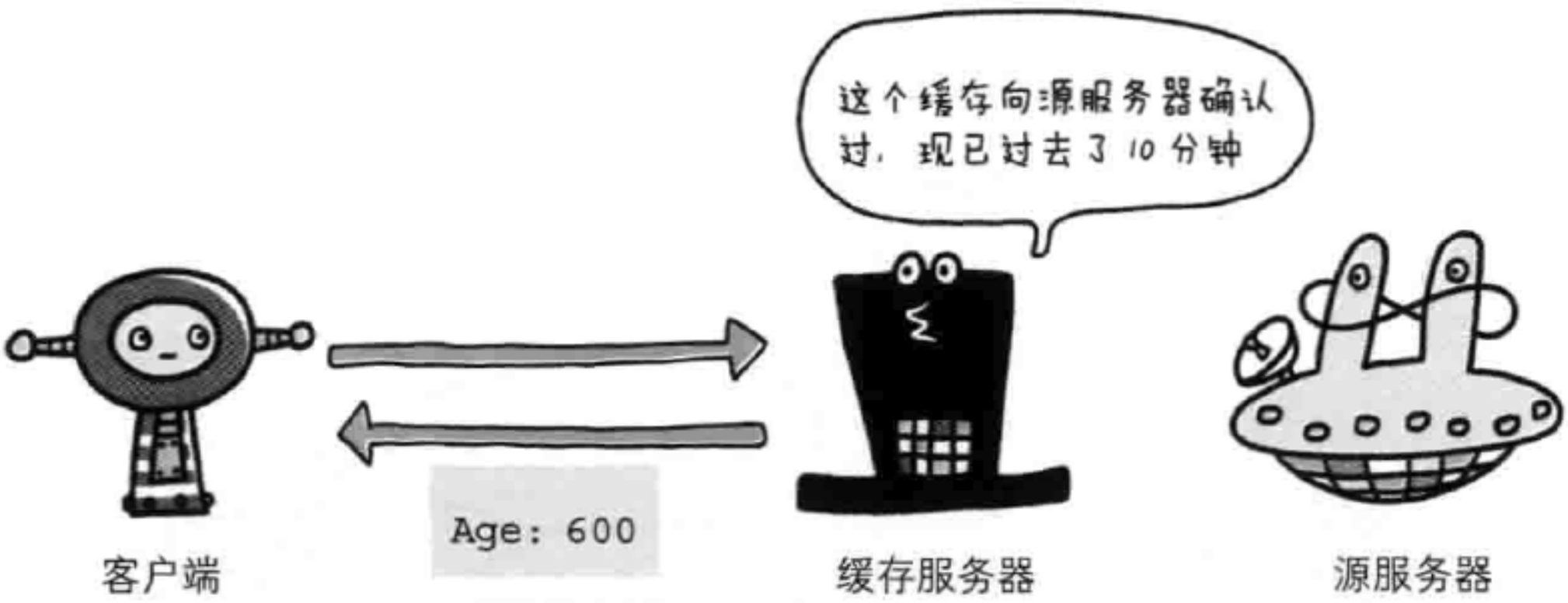
图：当不能处理范围请求时，Accept-Ranges: none

```
Accept-Ranges: bytes
```

首部字段 Accept-Ranges 是用来告知客户端服务器是否能处理范围请求，以指定获取服务器端某个部分的资源。

可指定的字段值有两种，可处理范围请求时指定其为 bytes，反之则指定其为 none。

6.5.2 Age

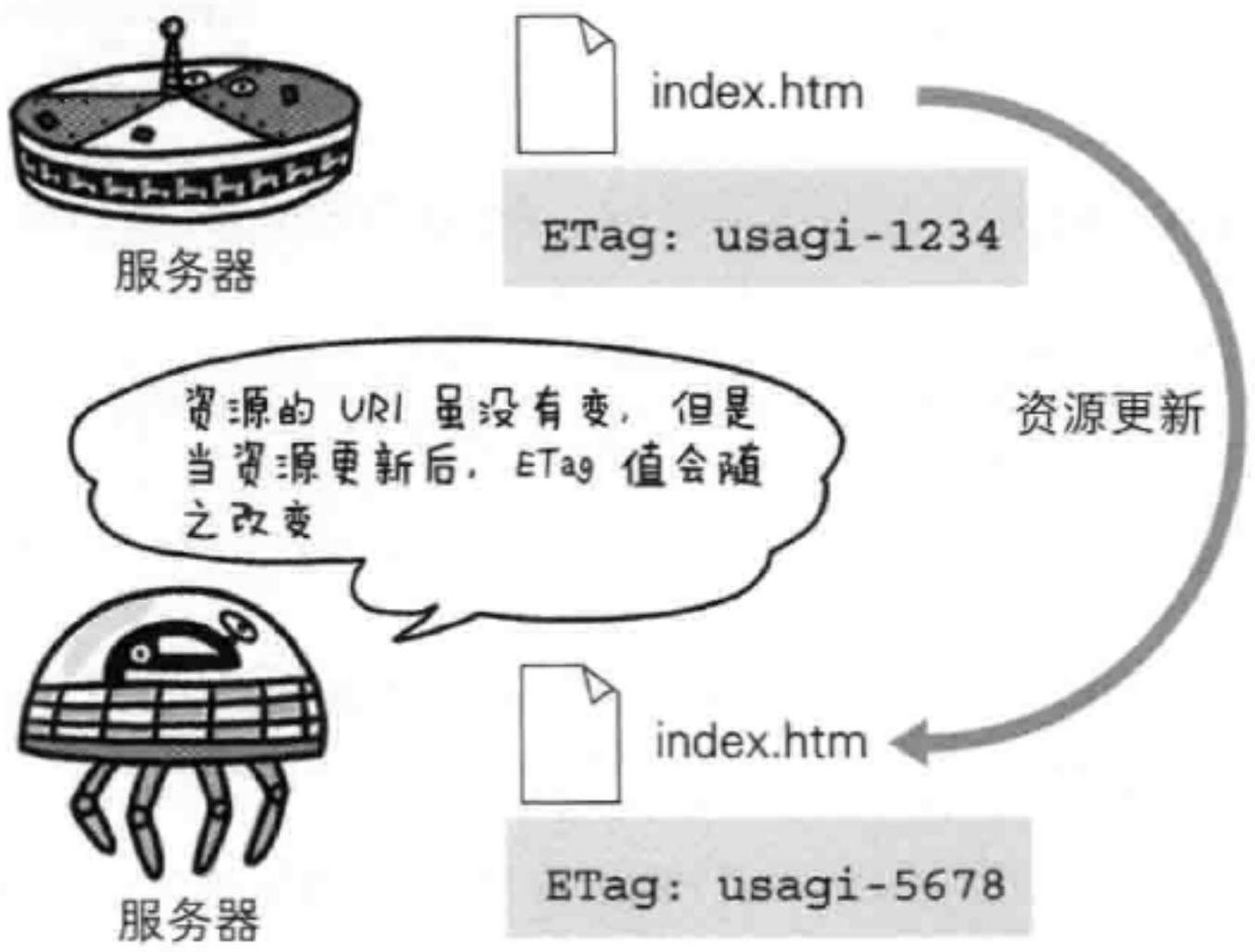



```
Age: 600
```

首部字段 Age 能告知客户端，源服务器在多久前创建了响应。字段值的单位为秒。

若创建该响应的服务器是缓存服务器，Age 值是指缓存后的响应再次发起认证到认证完成的时间值。代理创建响应时必须加上首部字段 Age。

6.5.3 ETag



```
ETag: "82e22293907ce725faf67773957acd12"
```

首部字段 ETag 能告知客户端实体标识。它是一种可将资源以字符串形式做唯一性标识的方式。服务器会为每份资源分配对应的 ETag 值。

另外，当资源更新时，ETag 值也需要更新。生成 ETag 值时，并没有统一的算法规则，而仅仅是由服务器来分配。



	中文版	英文版
		
服务器	`ETag: usagi-abcd`	`ETag: usagi-efgh`

资源被缓存时，就会被分配唯一性标识。例如，当使用中文版的浏览器访问 http：//www.google.com/ 时，就会返回中文版对应的资源，而使用英文版的浏览器访问时，则会返回英文版对应的资源。两者的 URI 是相同的，所以仅凭 URI 指定缓存的资源是相当困难的。若在下载过程中出现连接中断、再连接的情况，都会依照 ETag 值来指定资源。

强 ETag 值和弱 Tag 值

ETag 中有强 ETag 值和弱 ETag 值之分。

强 ETag 值

强 ETag 值，不论实体发生多么细微的变化都会改变其值。

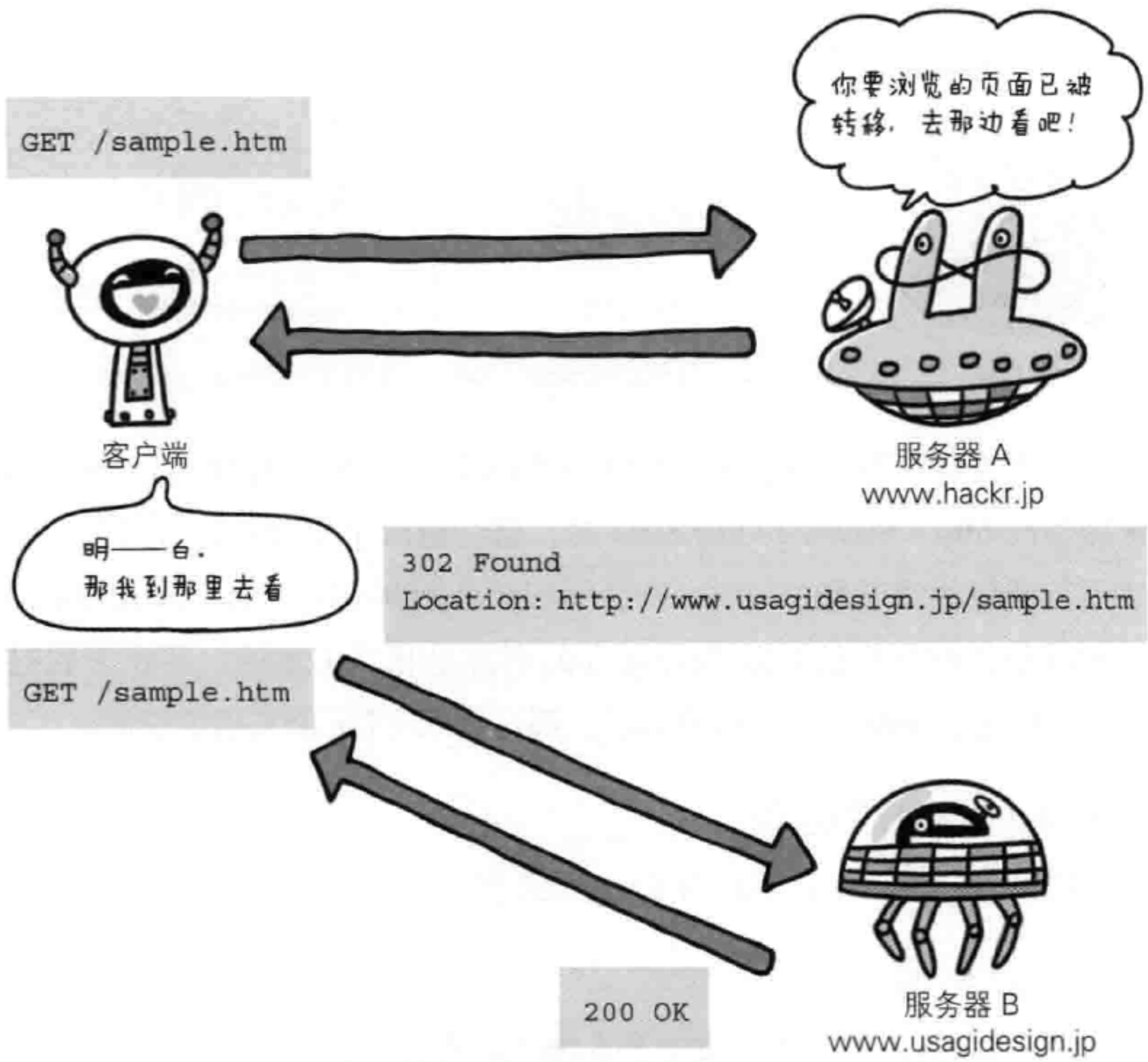
```
ETag: "usagi-1234"
```

弱 ETag 值

弱 ETag 值只用于提示资源是否相同。只有资源发生了根本改变，产生差异时才会改变 ETag 值。这时，会在字段值最开始处附加 W/。

```
ETag: W/"usagi-1234"
```

6.5.4 Location




```
Location: http://www.usagidesign.jp/sample.html
```

使用首部字段 Location 可以将响应接收方引导至某个与请求 URI 位置不同的资源。

基本上，该字段会配合 3xx：Redirection 的响应，提供重定向的 URI。

几乎所有的浏览器在接收到包含首部字段 Location 的响应后，都会强制性地尝试对已提示的重定向资源的访问。

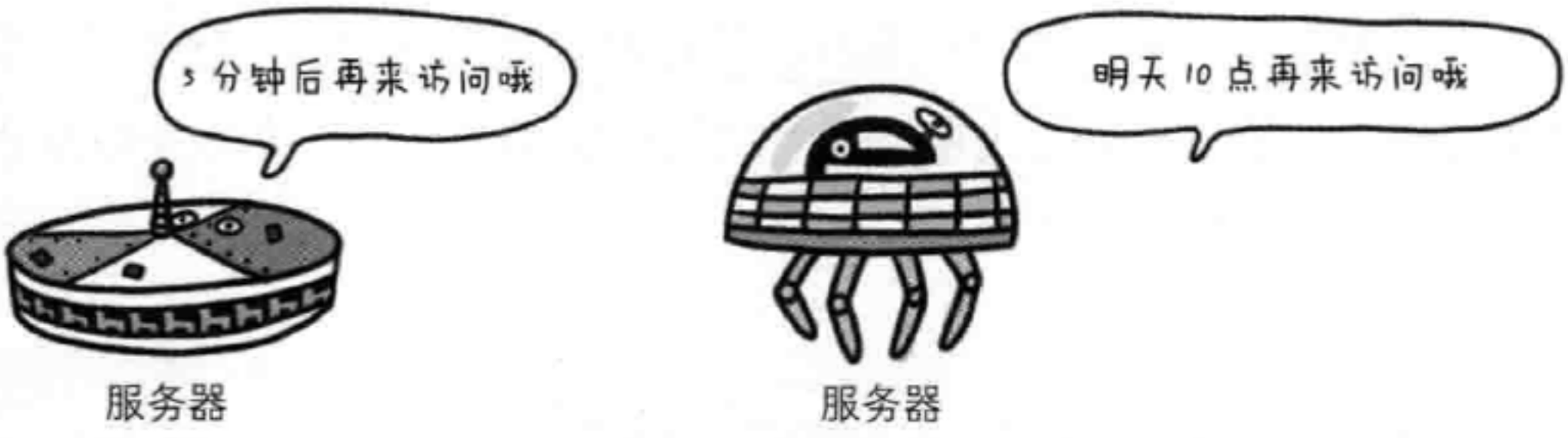
6.5.5 Proxy-Authenticate

```
Proxy-Authenticate: Basic realm="Usagidesign Auth"
```

首部字段 Proxy-Authenticate 会把由代理服务器所要求的认证信息发送给客户端。

它与客户端和服务器之间的 HTTP 访问认证的行为相似，不同之处在于其认证行为是在客户端与代理之间进行的。而客户端与服务器之间进行认证时，首部字段 WWW-Authorization 有着相同的作用。有关 HTTP 访问认证，后面的章节会再进行详尽阐述。

6.5.6 Retry-After

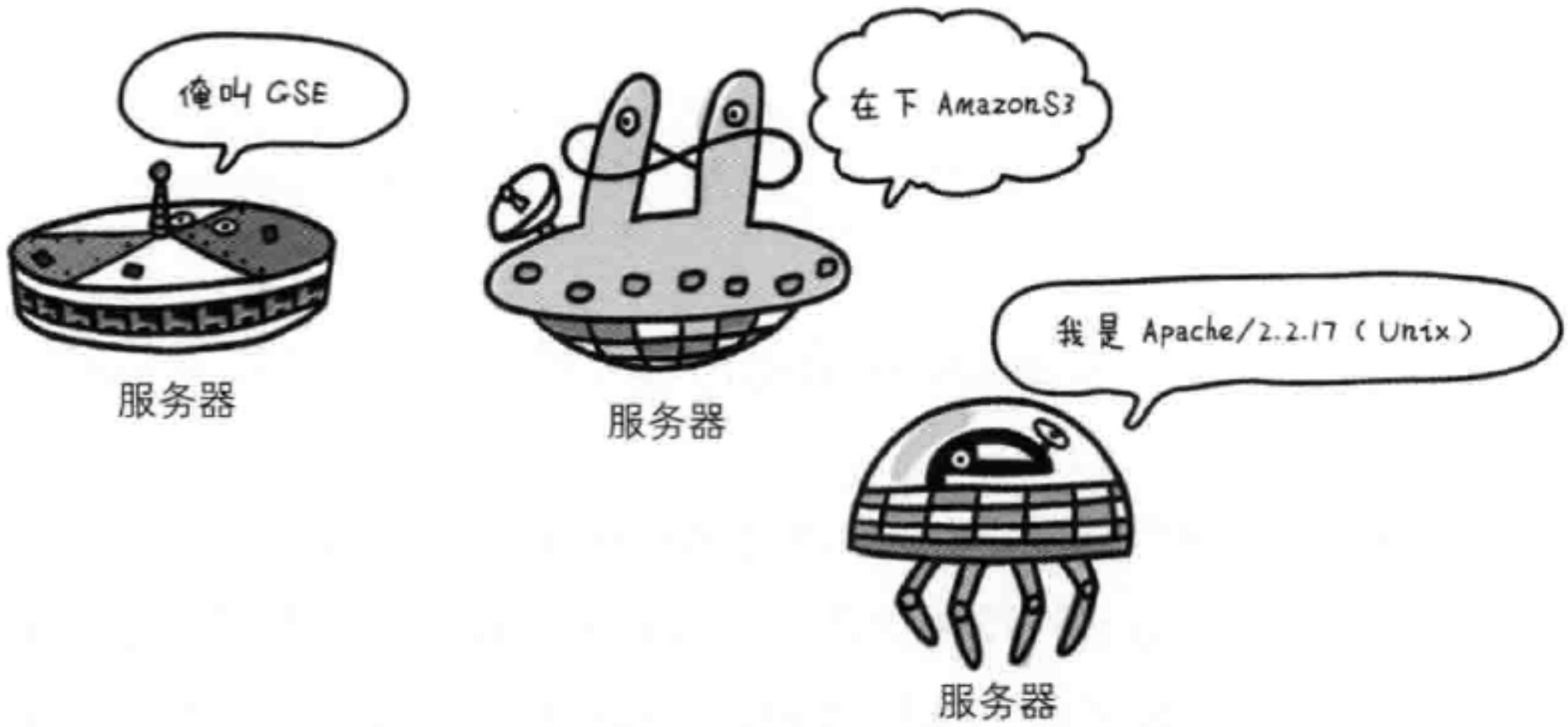



```
Retry-After: 120
```

首部字段 Retry-After 告知客户端应该在多久之后再次发送请求。主要配合状态码 503 Service Unavailable 响应，或 3xx Redirect 响应一起使用。

字段值可以指定为具体的日期时间（Wed, 04 Jul 2012 06：34：24 GMT 等格式），也可以是创建响应后的秒数。

6.5.7 Server

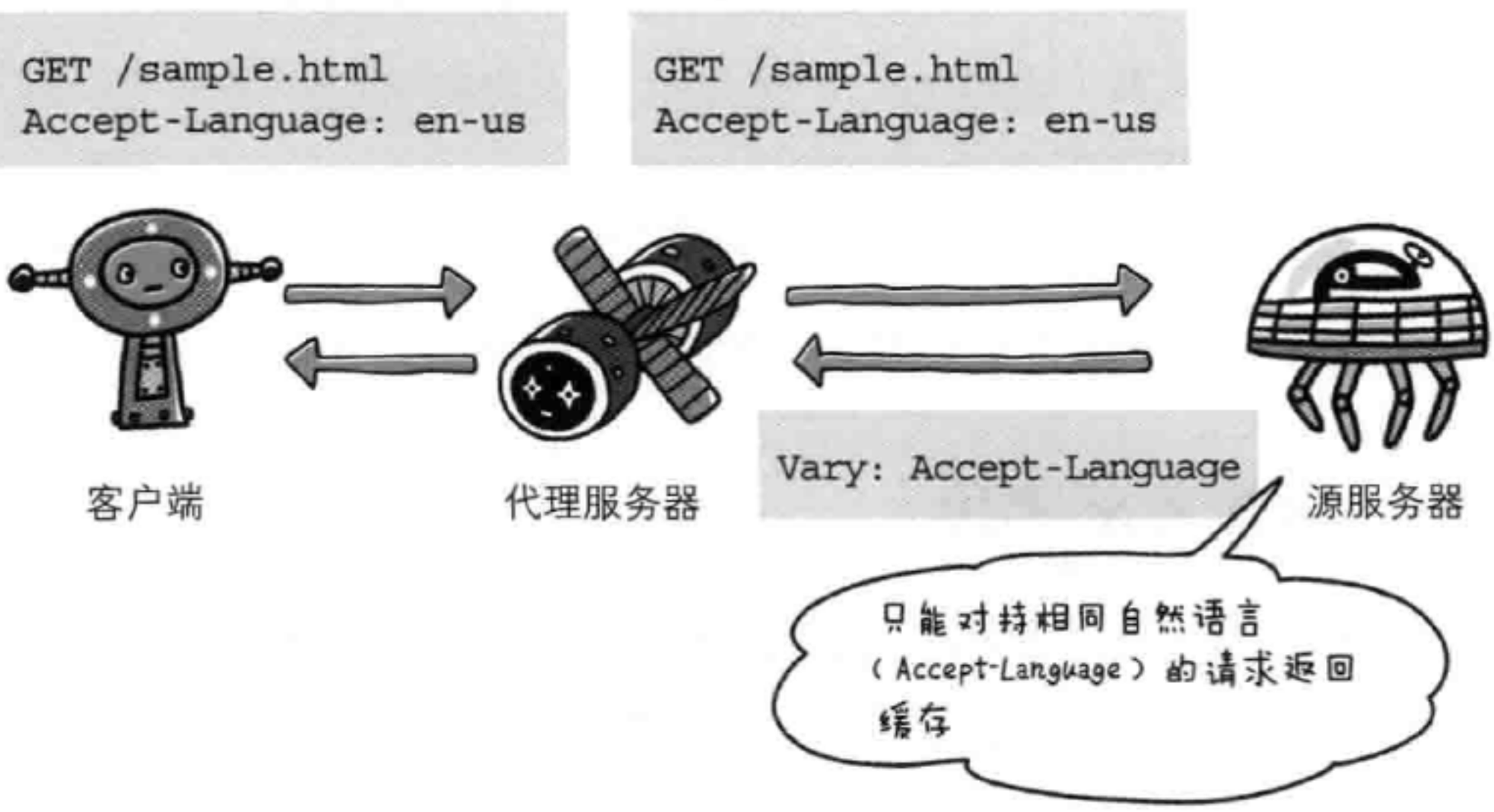



```
Server: Apache/2.2.17 (Unix)
```

首部字段 Server 告知客户端当前服务器上安装的 HTTP 服务器应用程序的信息。不单单会标出服务器上的软件应用名称，还有可能包括版本号和安装时启用的可选项。

```
Server: Apache/2.2.6 (Unix) PHP/5.2.5
```

6.5.8 Vary



图：当代理服务器接收到带有 Vary 首部字段指定获取资源的请求时，如果使用的 Accept-Language 字段的值相同，那么就直接从缓存返回响应。反之，则需要先从源服务器端获取资源后才能作为响应返回


```
Vary: Accept-Language
```

首部字段 Vary 可对缓存进行控制。源服务器会向代理服务器传达关于本地缓存使用方法的命令。

从代理服务器接收到源服务器返回包含 Vary 指定项的响应之后，若再要进行缓存，仅对请求中含有相同 Vary 指定首部字段的请求返回缓存。即使对相同资源发起请求，但由于 Vary 指定的首部字段不相同，因此必须要从源服务器重新获取资源。

6.5.9 WWW-Authenticate

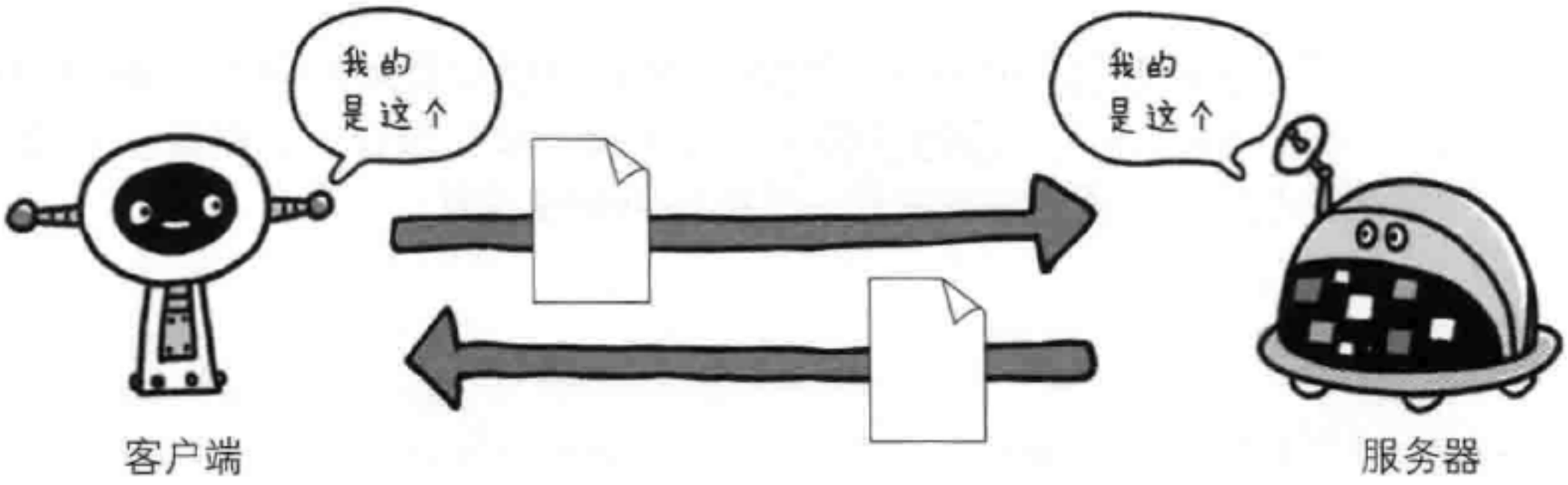
```
WWW-Authenticate: Basic realm="Usagidesign Auth"
```

首部字段 WWW-Authenticate 用于 HTTP 访问认证。它会告知客户端适用于访问请求 URI 所指定资源的认证方案（Basic 或是 Digest）和带参数提示的质询（challenge）。状态码 401 Unauthorized 响应中，肯定带有首部字段 WWW-Authenticate。

上述示例中，realm 字段的字符串是为了辨别请求 URI 指定资源所受到的保护策略。有关该首部，请参阅本章之后的内容。

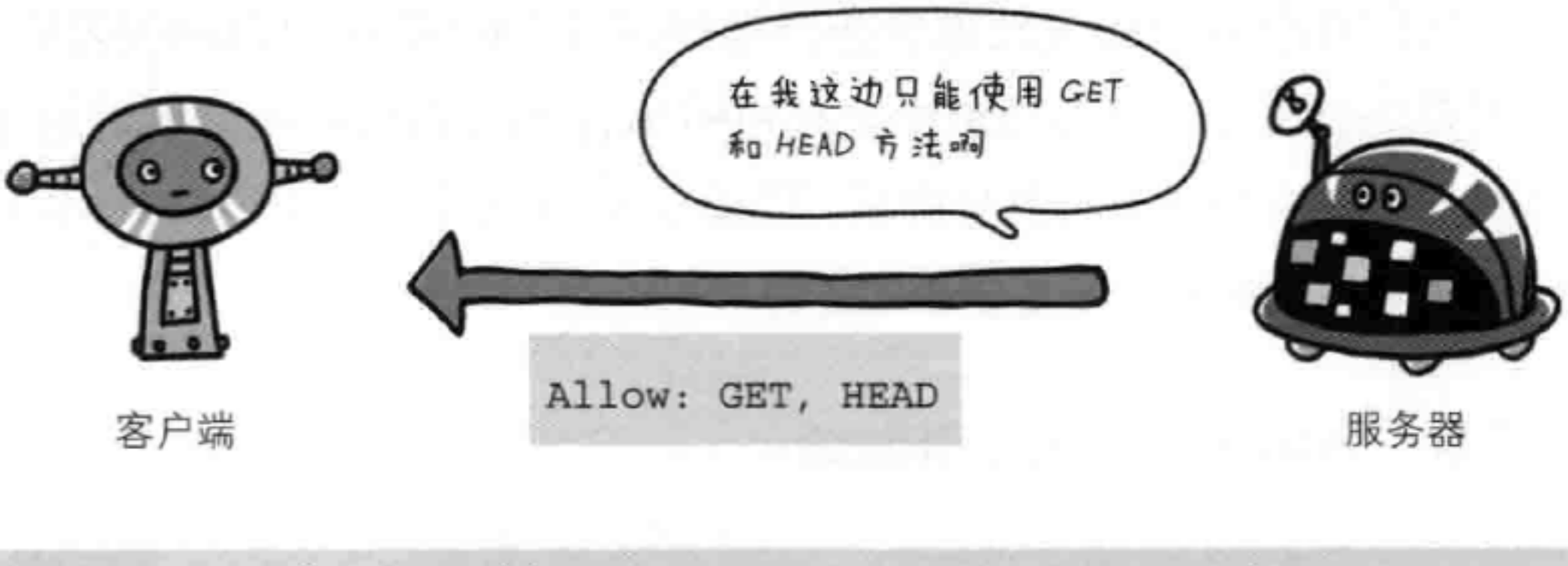
6.6 实体首部字段

实体首部字段是包含在请求报文和响应报文中的实体部分所使用的首部，用于补充内容的更新时间等与实体相关的信息。



图：在请求和响应两方的 HTTP 报文中都含有与实体相关的首部字段

6.6.1 Allow




```
Allow: GET, HEAD
```

首部字段 Allow 用于通知客户端能够支持 Request-URI 指定资源的所有 HTTP 方法。当服务器接收到不支持的 HTTP 方法时，会以状态码 405 Method Not Allowed 作为响应返回。与此同时，还会把所有能支持的 HTTP 方法写入首部字段 Allow 后返回。

6.6.2 Content-Encoding

```
Content-Encoding: gzip
```

首部字段 Content-Encoding 会告知客户端服务器对实体的主体部分选用的内容编码方式。内容编码是指在不丢失实体信息的前提下所进行的压缩。



服务器

主要采用以下 4 种内容编码的方式。（各方式的说明请参考 6.4.3 节 Accept-Encoding 首部字段）。

- gzip
- compress
- deflate
- identity

6.6.3 Content-Language



服务器

```
Content-Language: zh-CN
```

首部字段 Content-Language 会告知客户端，实体主体使用的自然语言（指中文或英文等语言）。

6.6.4 Content-Length



服务器

```
Content-Length: 15000
```

首部字段 Content-Length 表明了实体主体部分的大小（单位是字节）。对实体主体进行内容编码传输时，不能再使用 Content-Length 首部字段。由于实体主体大小的计算方法略微复杂，所以在此不再展开。读者若想一探究竟，可参考 RFC2616 的 4.4。

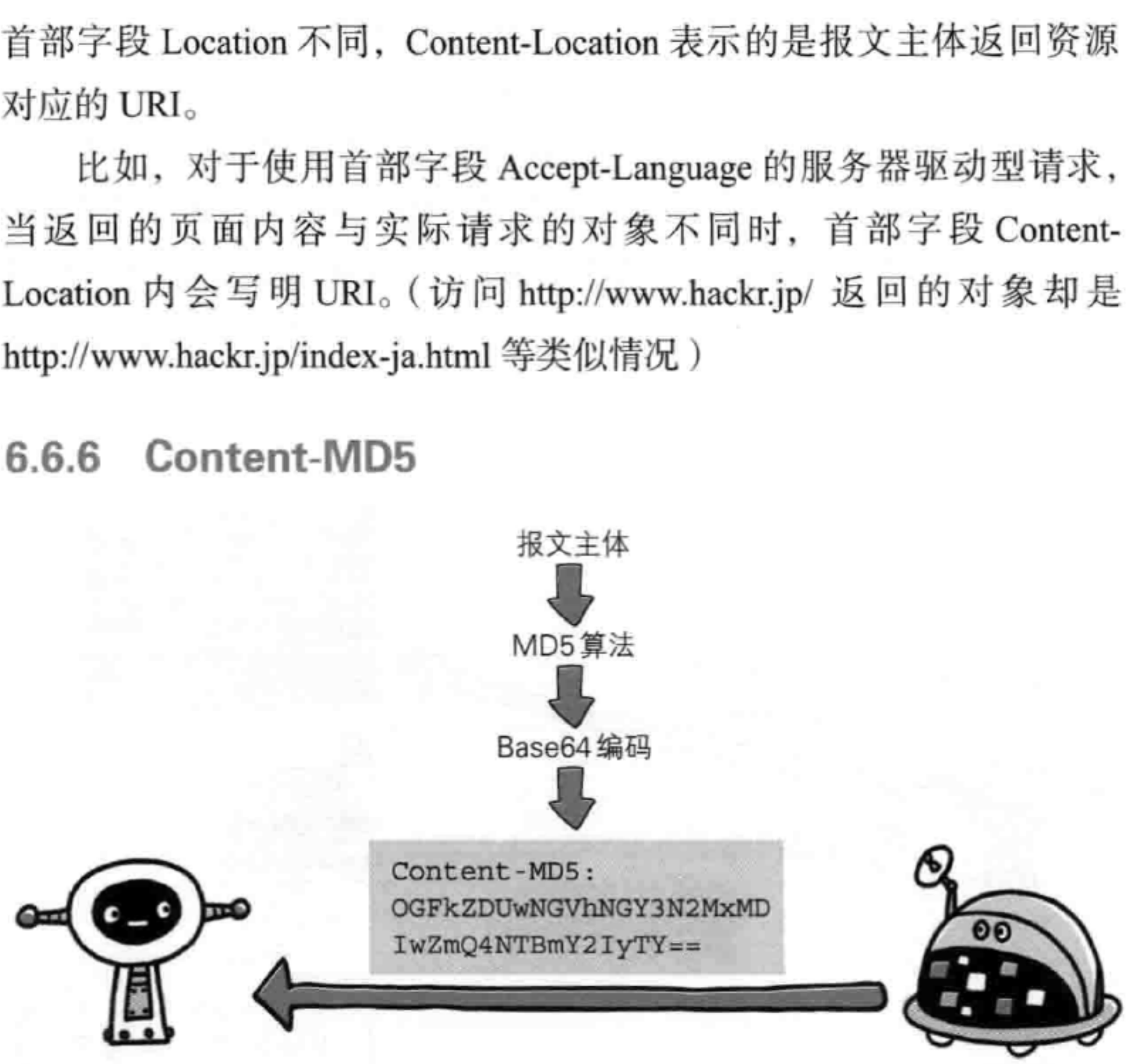
6.6.5 Content-Location

```
Content-Location: http://www.hackr.jp/index-ja.html
```

首部字段 Content-Location 给出与报文主体部分相对应的 URI。和首部字段 Location 不同，Content-Location 表示的是报文主体返回资源对应的 URI。

比如，对于使用首部字段 Accept-Language 的服务器驱动型请求，当返回的页面内容与实际请求的对象不同时，首部字段 Content-Location 内会写明 URI。（访问 http://www.hackr.jp/ 返回的对象却是 http://www.hackr.jp/index-ja.html 等类似情况）

6.6.6 Content-MD5



图：客户端会对接收的报文主体执行相同的 MD5 算法，然后与首部字段 Content-MD5 的字段值比较

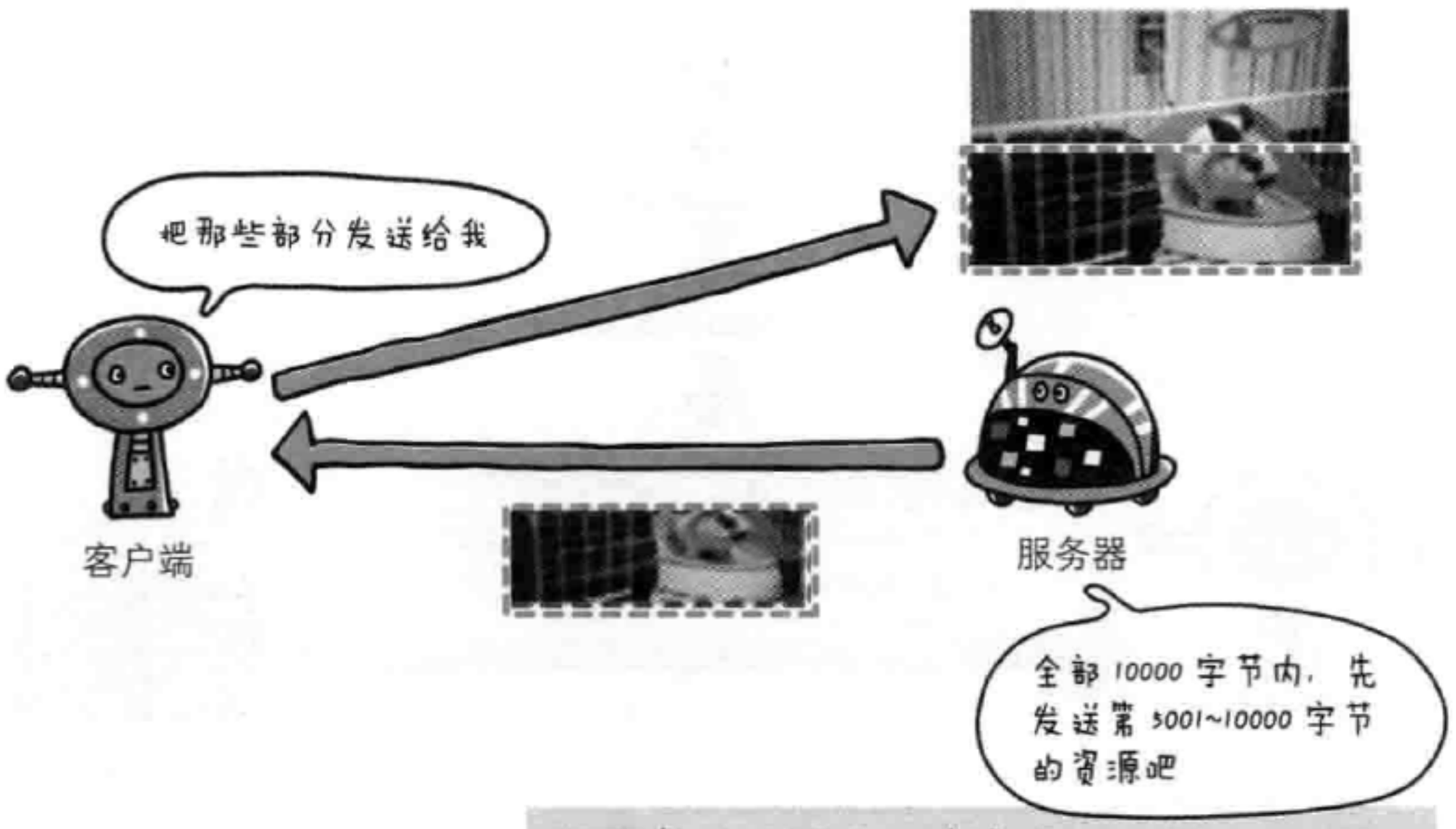
```
Content-MD5: OGFkZDUwNGVhNGY3N2MxMDIwZmQ4NTBmY2IyTY==
```

首部字段 Content-MD5 是一串由 MD5 算法生成的值，其目的在于检查报文主体在传输过程中是否保持完整，以及确认传输到达。

对报文主体执行 MD5 算法获得的 128 位二进制数，再通过 Base64 编码后将结果写入 Content-MD5 字段值。由于 HTTP 首部无法记录二进制值，所以要通过 Base64 编码处理。为确保报文的有效性，作为接收方的客户端会对报文主体再执行一次相同的 MD5 算法。计算出的值与字段值作比较后，即可判断出报文主体的准确性。

采用这种方法，对内容上的偶发性改变是无从查证的，也无法检测出恶意篡改。其中一个原因在于，内容如果能够被篡改，那么同时意味着 Content-MD5 也可重新计算然后被篡改。所以处在接收阶段的客户端是无法意识到报文主体以及首部字段 Content-MD5 是已经被篡改过的。

6.6.7 Content-Range




```
HTTP/1.1 206 Partial Content
Date: Wed, 04 Jul 2012 07:28:03 GMT
Content-Range: bytes 5001-10000/10000
Content-Length: 5000
Content-Type: image/jpeg
```

```
Content-Range: bytes 5001-10000/10000
```

针对范围请求，返回响应时使用的首部字段 Content-Range，能告知客户端作为响应返回的实体的哪个部分符合范围请求。字段值以字节为单位，表示当前发送部分及整个实体大小。

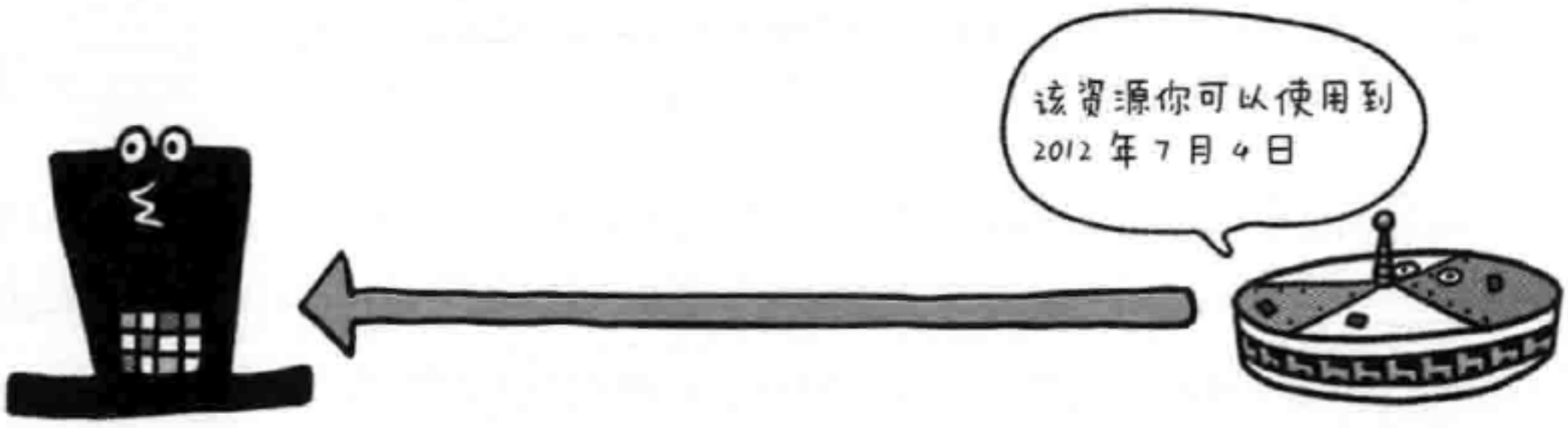
6.6.8 Content-Type

```
Content-Type: text/html; charset=UTF-8
```

首部字段 Content-Type 说明了实体主体内对象的媒体类型。和首部字段 Accept 一样，字段值用 type/subtype 形式赋值。

参数 charset 使用 iso-8859-1 或 euc-jp 等字符集进行赋值。

6.6.9 Expires




```
Expires: Wed, 04 Jul 2012 08:26:05 GMT
```

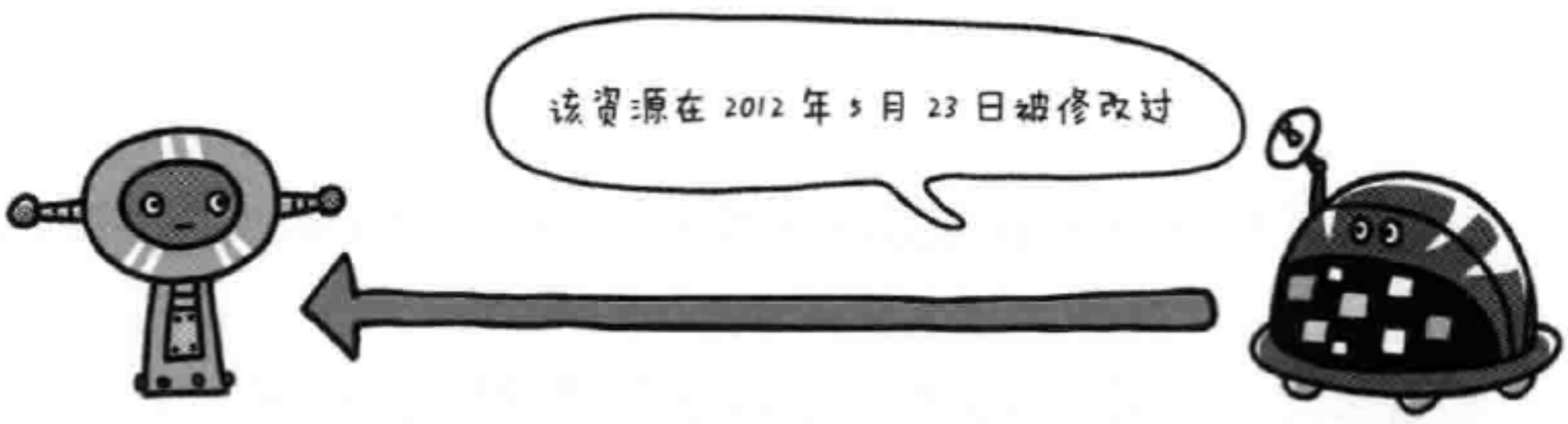
首部字段 Expires 会将资源失效的日期告知客户端。缓存服务器在接收到含有首部字段 Expires 的响应后，会以缓存来应答请求，在

Expires 字段值指定的时间之前，响应的副本会一直被保存。当超过指定的时间后，缓存服务器在请求发送过来时，会转向源服务器请求资源。

源服务器不希望缓存服务器对资源缓存时，最好在 Expires 字段内写入与首部字段 Date 相同的时间值。

但是，当首部字段 Cache-Control 有指定 max-age 指令时，比起首部字段 Expires，会优先处理 max-age 指令。

6.6.10 Last-Modified




```
Last-Modified: Wed, 23 May 2012 09:59:55 GMT
```

首部字段 Last-Modified 指明资源最终修改的时间。一般来说，这个值就是 Request-URI 指定资源被修改的时间。但类似使用 CGI 脚本进行动态数据处理时，该值有可能会变成数据最终修改时的时间。

6.7 为 Cookie 服务的首部字段

管理服务器与客户端之间状态的 Cookie，虽然没有被编入标准化 HTTP/1.1 的 RFC2616 中，但在 Web 网站方面得到了广泛的应用。

Cookie 的工作机制是用户识别及状态管理。Web 网站为了管理用户的状态会通过 Web 浏览器，把一些数据临时写入用户的计算机内。接

着当用户访问该Web网站时，可通过通信方式取回之前发放的Cookie。

调用Cookie时，由于可校验Cookie的有效期，以及发送方的域、路径、协议等信息，所以正规发布的Cookie内的数据不会因来自其他Web站点和攻击者的攻击而泄露。

至2013年5月，Cookie的规格标准文档有以下4种。

由网景公司颁布的规格标准

网景通信公司设计并开发了Cookie，并制定相关的规格标准。1994年前后，Cookie正式应用在网景浏览器中。目前最为普及的Cookie方式也是以此为基准的。

RFC2109

某企业尝试以独立技术对Cookie规格进行标准化统筹。原本的意图是想和网景公司制定的标准交互应用，可惜发生了微妙的差异。现在该标准已淡出了人们的视线。

RFC2965

为终结Internet Explorer浏览器与Netscape Navigator的标准差异而导致的浏览器战争，RFC2965内定义了新的HTTP首部Set-Cookie2和Cookie2。可事实上，它们几乎没怎么投入使用。

RFC6265

将网景公司制定的标准作为业界事实标准（De facto standard），重新定义Cookie标准后的产物。

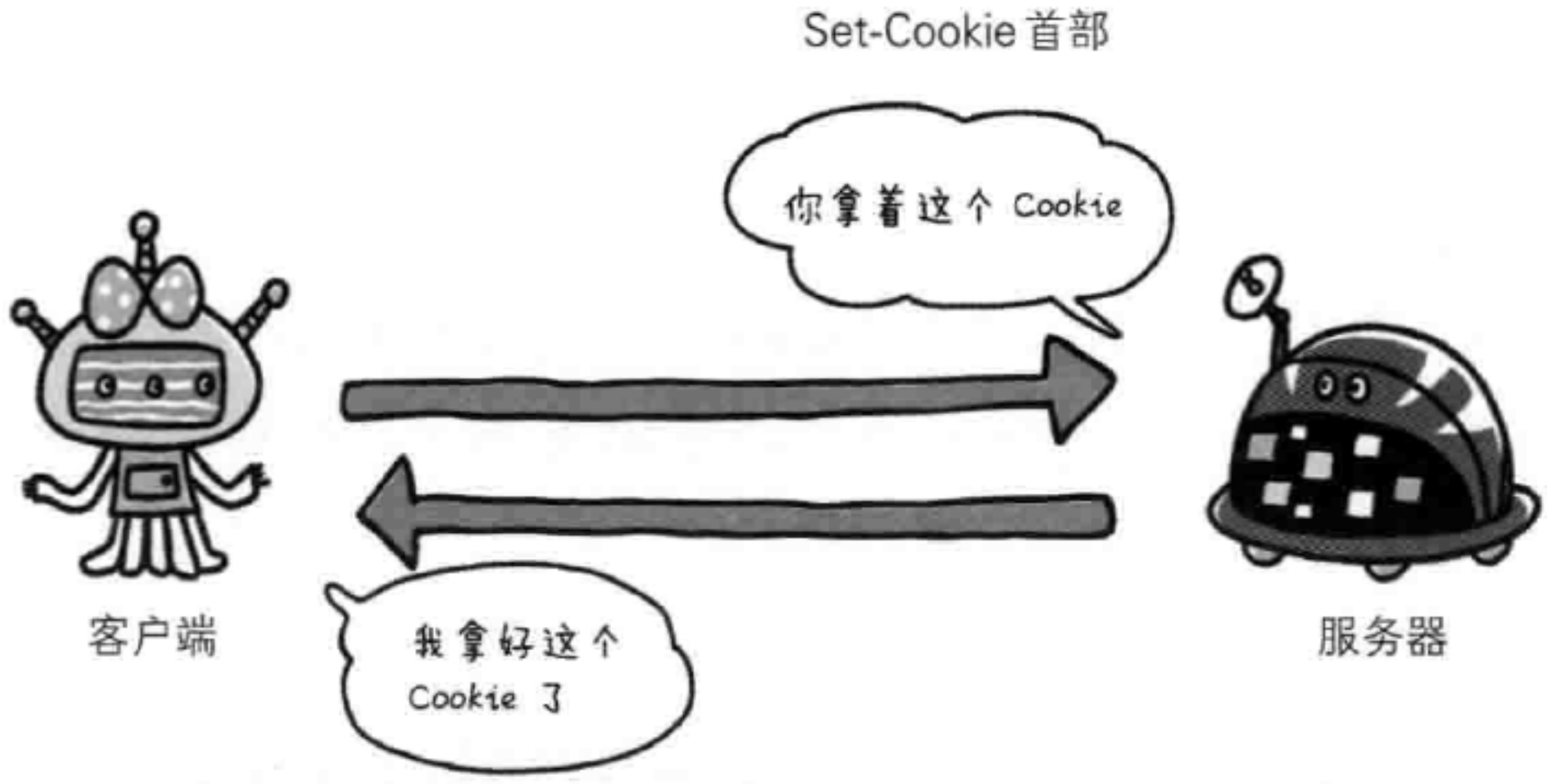
目前使用最广泛的Cookie标准却不是RFC中定义的任何一个。而是在网景公司制定的标准上进行扩展后的产物。

本节接下来就对目前使用最为广泛普及的标准进行说明。

下面的表格内列举了与Cookie有关的首部字段。

表 6-8：为 Cookie 服务的首部字段

首部字段名	说明	首部类型
Set-Cookie	开始状态管理所使用的Cookie信息	响应首部字段
Cookie	服务器接收到的Cookie信息	请求首部字段



6.7.1 Set-Cookie

```
Set-Cookie: status=enable; expires=Tue, 05 Jul 2011 07:26:31 GMT; ⇒
path=/; domain=.hackr.jp;
```

当服务器准备开始管理客户端的状态时，会事先告知各种信息。

下面的表格列举了 Set-Cookie 的字段值。

表 6-9：Set-Cookie 字段的属性

属性	说明
NAME=VALUE	赋予Cookie的名称和其值（必需项）
expires=DATE	Cookie的有效期（若不明确指定则默认为浏览器关闭前为止）
path=PATH	将服务器上的文件目录作为Cookie的适用对象（若不指定则默认为文档所在的文件目录）
domain=域名	作为Cookie适用对象的域名（若不指定则默认为创建Cookie的服务器的域名）

（续）

属性	说明
Secure	仅在HTTPS安全通信时才会发送Cookie
HttpOnly	加以限制，使Cookie不能被JavaScript脚本访问

expires 属性

Cookie 的 expires 属性指定浏览器可发送 Cookie 的有效期。

当省略 expires 属性时，其有效期仅限于维持浏览器会话（Session）时间段内。这通常限于浏览器应用程序被关闭之前。

另外，一旦 Cookie 从服务器端发送至客户端，服务器端就不存在可以显式删除 Cookie 的方法。但可通过覆盖已过期的 Cookie，实现对客户端 Cookie 的实质性删除操作。

path 属性

Cookie 的 path 属性可用于限制指定 Cookie 的发送范围的文件目录。不过另有办法可避开这项限制，看来对其作为安全机制的效果不能抱有期待。

domain 属性

通过 Cookie 的 domain 属性指定的域名可做到与结尾匹配一致。比如，当指定 example.com 后，除 example.com 以外，www.example.com 或 www2.example.com 等都可以发送 Cookie。

因此，除了针对具体指定的多个域名发送 Cookie 之外，不指定 domain 属性显得更安全。

secure 属性

Cookie 的 secure 属性用于限制 Web 页面仅在 HTTPS 安全连接时，才可以发送 Cookie。

发送 Cookie 时，指定 secure 属性的方法如下所示。

```
Set-Cookie: name=value; secure
```

以上例子仅当在 https：//www.example.com/（HTTPS）安全连接的情况下才会进行 Cookie 的回收。也就是说，即使域名相同，http://www.example.com/（HTTP）也不会发生 Cookie 回收行为。

当省略 secure 属性时，不论 HTTP 还是 HTTPS，都会对 Cookie 进行回收。

HttpOnly 属性

Cookie 的 HttpOnly 属性是 Cookie 的扩展功能，它使 JavaScript 脚本无法获得 Cookie。其主要目的为防止跨站脚本攻击（Cross-site scripting，XSS）对 Cookie 的信息窃取。

发送指定 HttpOnly 属性的 Cookie 的方法如下所示。

```
Set-Cookie: name=value; HttpOnly
```

通过上述设置，通常从 Web 页面内还可以对 Cookie 进行读取操作。但使用 JavaScript 的 document.cookie 就无法读取附加 HttpOnly 属性后的 Cookie 的内容了。因此，也就无法在 XSS 中利用 JavaScript 劫持 Cookie 了。

虽然是独立的扩展功能，但 Internet Explorer 6 SP1 以上版本等当下的主流浏览器都已经支持该扩展了。另外顺带一提，该扩展并非是为了防止 XSS 而开发的。

6.7.2 Cookie

```
Cookie: status=enable
```

首部字段 Cookie 会告知服务器，当客户端想获得 HTTP 状态管理支持时，就会在请求中包含从服务器接收到的 Cookie。接收到多个 Cookie 时，同样可以以多个 Cookie 形式发送。

6.8　其他首部字段

HTTP 首部字段是可以自行扩展的。所以在 Web 服务器和浏览器的应用上，会出现各种非标准的首部字段。

接下来，我们就一些最为常用的首部字段进行说明。

- X-Frame-Options
- X-XSS-Protection
- DNT
- P3P

6.8.1　X-Frame-Options

```
X-Frame-Options: DENY
```

首部字段 X-Frame-Options 属于 HTTP 响应首部，用于控制网站内容在其他 Web 网站的 Frame 标签内的显示问题。其主要目的是为了防止点击劫持（clickjacking）攻击。

首部字段 X-Frame-Options 有以下两个可指定的字段值。

- DENY：拒绝
- SAMEORIGIN：仅同源域名下的页面（Top-level-browsing-context）匹配时许可。（比如，当指定 http://hackr.jp/sample.html 页面为 SAMEORIGIN 时，那么 hackr.jp 上所有页面的 frame 都被允许可加载该页面，而 example.com 等其他域名的页面就不行了）

支持该首部字段的浏览器有：Internet Explorer 8、Firefox 3.6.9+、Chrome 4.1.249.1042+、Safari 4+ 和 Opera 10.50+ 等。现在主流的浏览器都已经支持。

能在所有的 Web 服务器端预先设定好 X-Frame-Options 字段值是最理想的状态。

对 apache2.conf 的配置实例

```
<IfModule mod_headers.c>
    Header append X-FRAME-OPTIONS "SAMEORIGIN"
</IfModule>
```

6.8.2 X-XSS-Protection

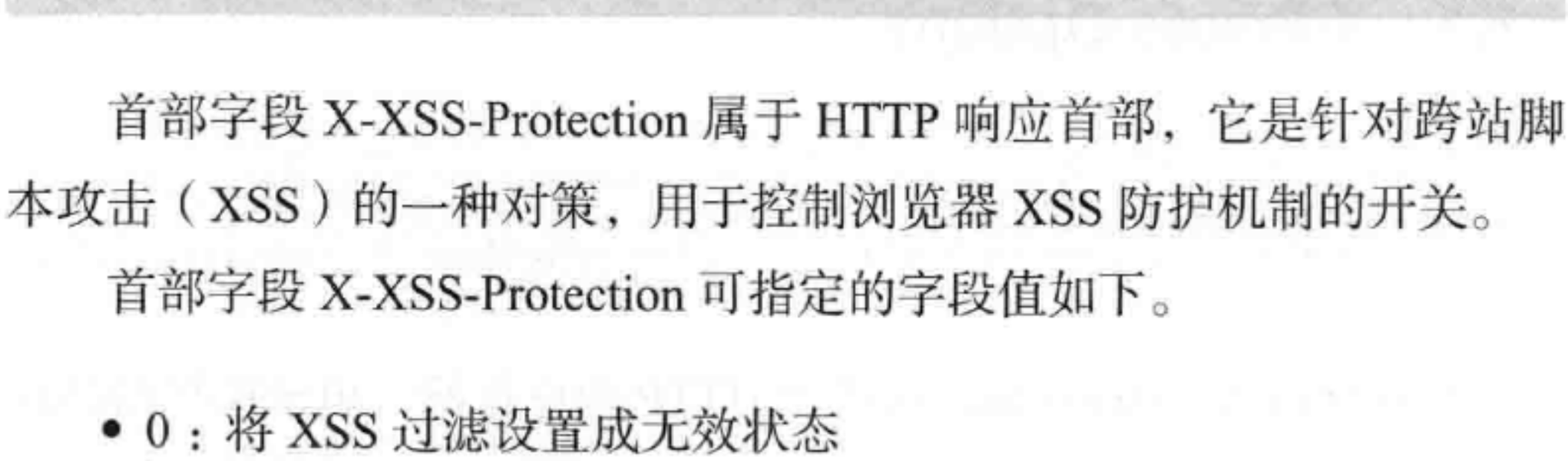
```
X-XSS-Protection: 1
```


首部字段 X-XSS-Protection 属于 HTTP 响应首部，它是针对跨站脚本攻击（XSS）的一种对策，用于控制浏览器 XSS 防护机制的开关。

首部字段 X-XSS-Protection 可指定的字段值如下。

- 0：将 XSS 过滤设置成无效状态
- 1：将 XSS 过滤设置成有效状态

6.8.3 DNT



```
DNT: 1
```

首部字段 DNT 属于 HTTP 请求首部，其中 DNT 是 Do Not Track 的简称，意为拒绝个人信息被收集，是表示拒绝被精准广告追踪的一种方法。

首部字段 DNT 可指定的字段值如下。

- 0：同意被追踪
- 1：拒绝被追踪

由于首部字段 DNT 的功能具备有效性，所以 Web 服务器需要对 DNT 做对应的支持。

6.8.4 P3P

```
P3P: CP="CAO DSP LAW CURa ADMa DEVa TAIa PSAa PSDa ⇒
IVAa IVDa OUR BUS IND UNI COM NAV INT"
```

首部字段 P3P 属于 HTTP 相应首部，通过利用 P3P（The Platform for Privacy Preferences，在线隐私偏好平台）技术，可以让 Web 网站上的个人隐私变成一种仅供程序可理解的形式，以达到保护用户隐私的目的。

要进行 P3P 的设定，需按以下操作步骤进行。

步骤 1： 创建 P3P 隐私

步骤 2： 创建 P3P 隐私对照文件后，保存命名在 /w3c/p3p.xml

步骤 3： 从 P3P 隐私中新建 Compact policies 后，输出到 HTTP 响应中

有关 P3P 的详细规范标准请参看下方链接。

- The Platform for Privacy Preferences 1.0（P3P1.0）Specification

  http://www.w3.org/TR/P3P/

协议中对 X- 前缀的废除

在 HTTP 等多种协议中，通过给非标准参数加上前缀 X-，来区别于标准参数，并使那些非标准的参数作为扩展变成可能。但是这种简单粗暴的做法有百害而无一益，因此在“RFC 6648 - Deprecating the "X-" Prefix and Similar Constructs in Application Protocols”中提议停止该做法。

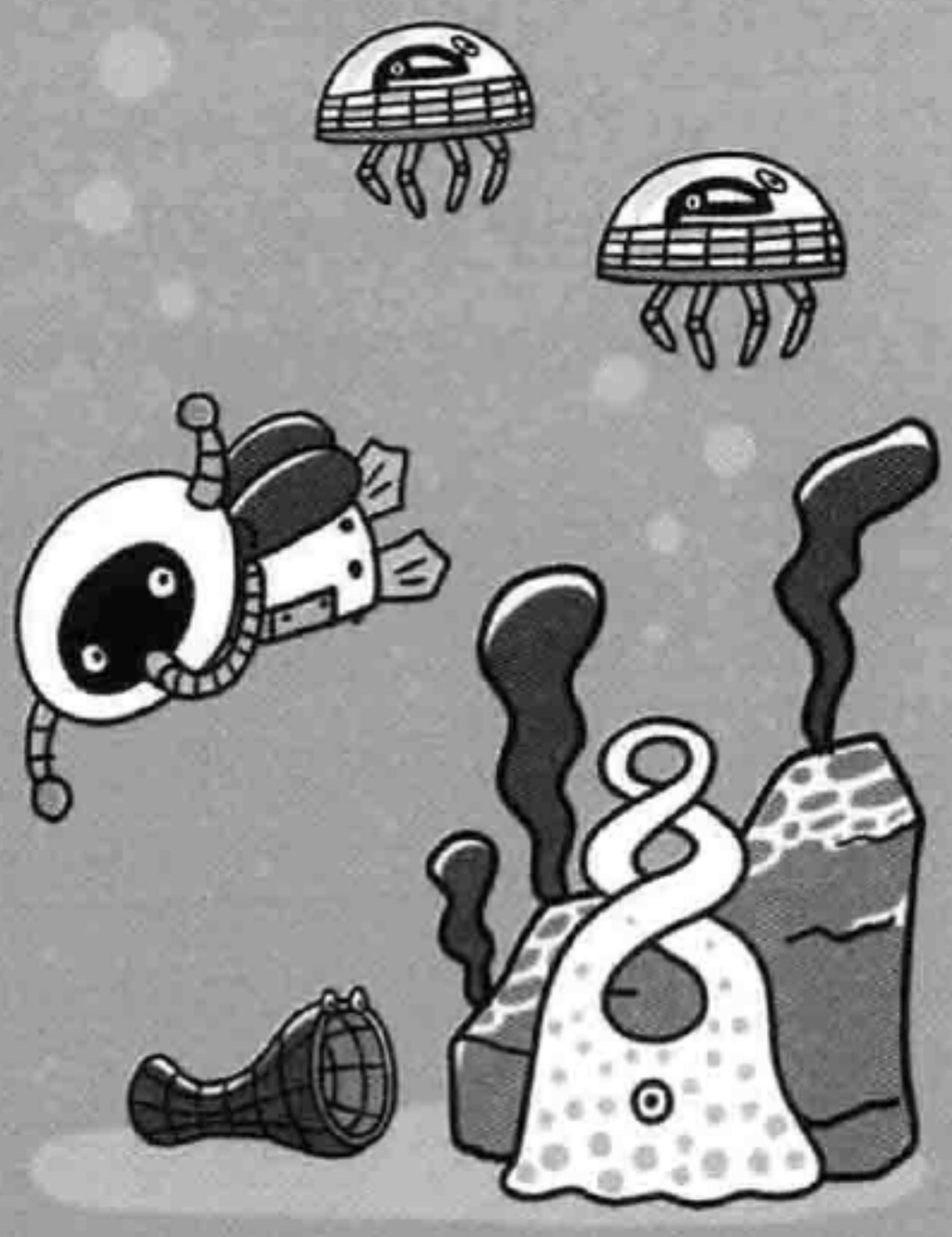
然而，对已经在使用中的 X- 前缀来说，不应该要求其变更。

Chapter7

第7章

确保Web安全的HTTPS

在HTTP协议中有可能存在信息窃听或身份伪装等安全问题。使用HTTPS通信机制可以有效地防止这些问题。本章我们就了解一下HTTPS。



7.1 HTTP 的缺点

到现在为止，我们已了解到 HTTP 具有相当优秀和方便的一面，然而 HTTP 并非只有好的一面，事物皆具两面性，它也是有不足之处的。HTTP 主要有这些不足，例举如下。

- 通信使用明文（不加密），内容可能会被窃听
- 不验证通信方的身份，因此有可能遭遇伪装
- 无法证明报文的完整性，所以有可能已遭篡改

这些问题不仅在 HTTP 上出现，其他未加密的协议中也会存在这类问题。

除此之外，HTTP 本身还有很多缺点。而且，还有像某些特定的 Web 服务器和特定的 Web 浏览器在实际应用中存在的不足（也可以说成是脆弱性或安全漏洞），另外，用 Java 和 PHP 等编程语言开发的 Web 应用也可能存在安全漏洞。

7.1.1 通信使用明文可能会被窃听

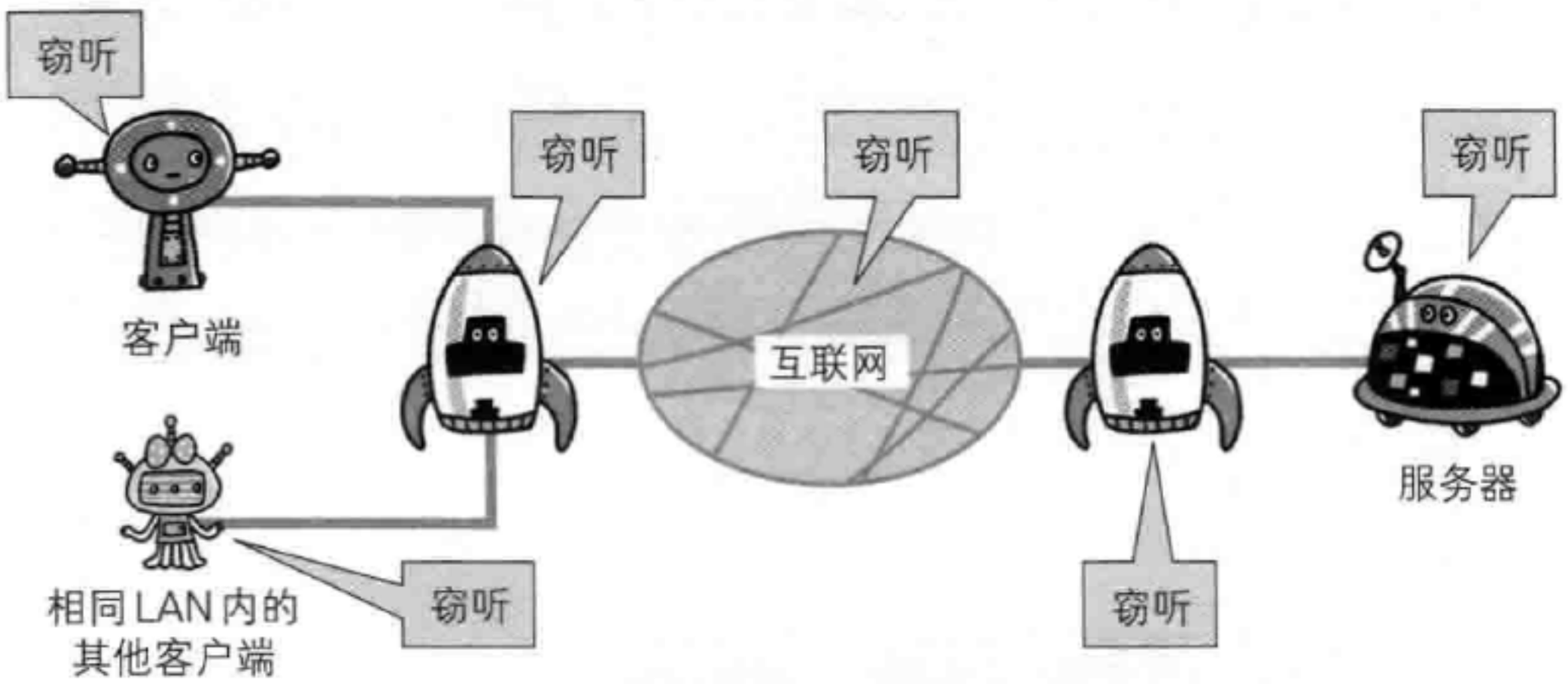
由于 HTTP 本身不具备加密的功能，所以也无法做到对通信整体（使用 HTTP 协议通信的请求和响应的内容）进行加密。即，HTTP 报文使用明文（指未经过加密的报文）方式发送。

■TCP/IP是可能被窃听的网络

如果要问为什么通信时不加密是一个缺点，这是因为，按 TCP/IP 协议族的工作机制，通信内容在所有的通信线路上都有可能遭到窥视。

所谓互联网，是由能连通到全世界的网络组成的。无论世界哪个角落的服务器在和客户端通信时，在此通信线路上的某些网络设备、光缆、计算机等都不可能是个人的私有物，所以不排除某个环节中会遭到恶意窥视行为。

即使已经过加密处理的通信，也会被窥视到通信内容，这点和未加密的通信是相同的。只是说如果通信经过加密，就有可能让人无法破解报文信息的含义，但加密处理后的报文信息本身还是会被看到的。

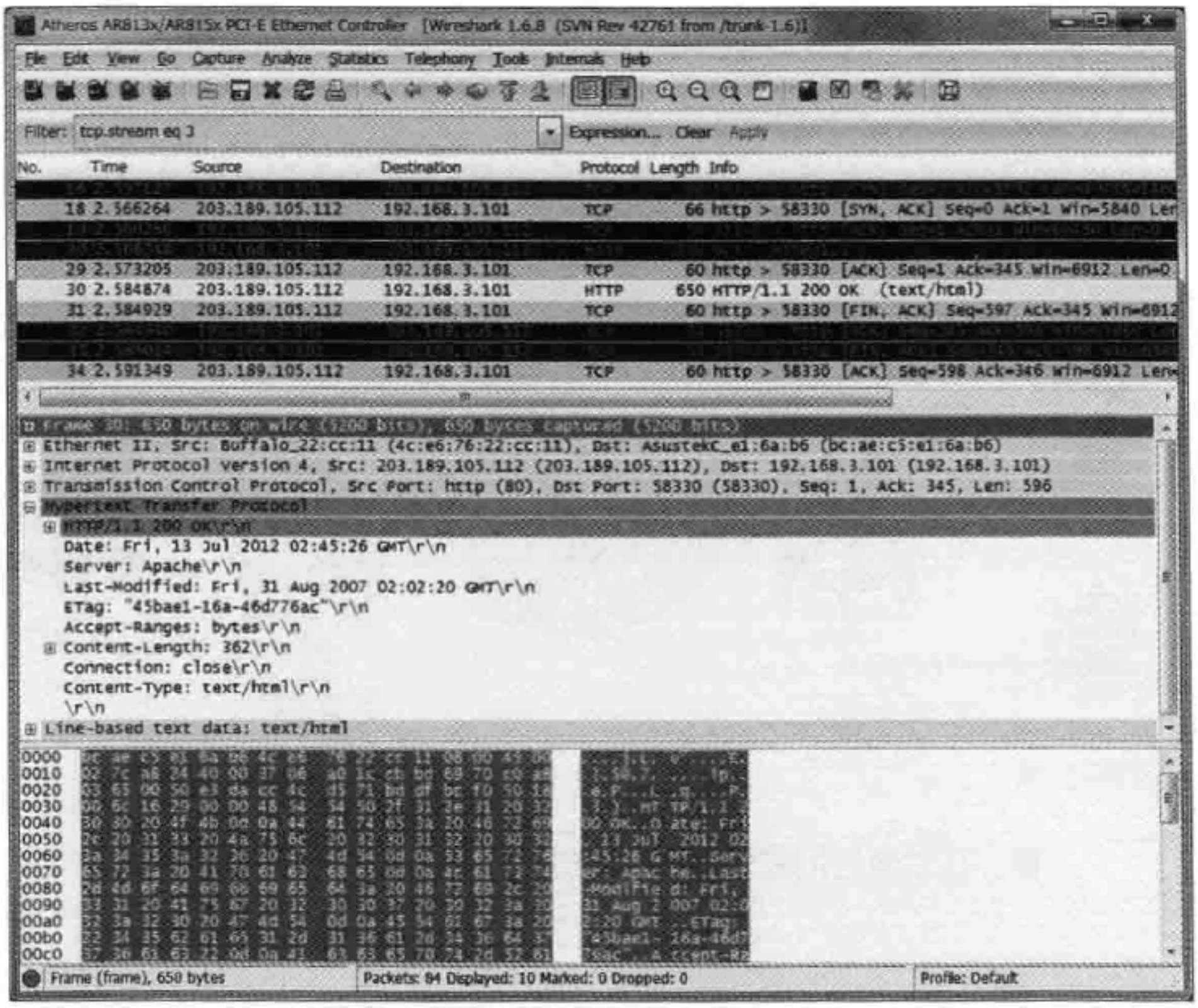


图：互联网上的任何角落都存在通信内容被窃听的风险

窃听相同段上的通信并非难事。只需要收集在互联网上流动的数据包（帧）就行了。对于收集来的数据包的解析工作，可交给那些抓包（Packet Capture）或嗅探器（Sniffer）工具。

下面的图片示例就是被广泛使用的抓包工具 Wireshark。它可以获取 HTTP 协议的请求和响应的内容，并对其进行解析。

像使用 GET 方法发送请求、响应返回了 200 OK，查看 HTTP 响应报文的全部内容等一系列的事情都可以做到。



图：Wireshark（http://www.wireshark.org/）

■加密处理防止被窃听

在目前大家正在研究的如何防止窃听保护信息的几种对策中，最为普及的就是加密技术。加密的对象可以有这么几个。

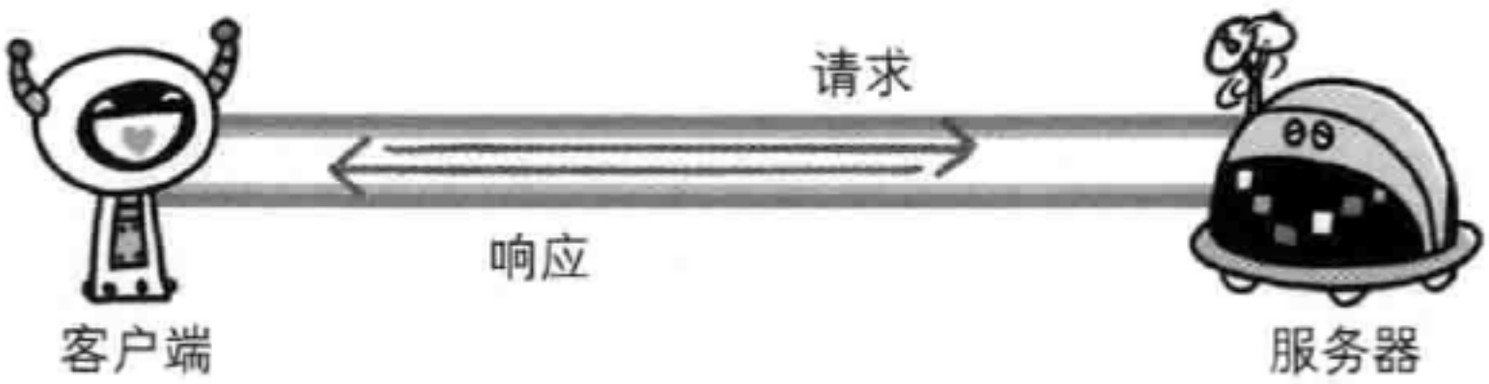
通信的加密

一种方式就是将通信加密。HTTP 协议中没有加密机制，但可以通过和 SSL（Secure Socket Layer，安全套接层）或 TLS（Transport Layer Security，安全层传输协议）的组合使用，加密 HTTP 的通信内容。

用 SSL 建立安全通信线路之后，就可以在这条线路上进行 HTTP 通信了。与 SSL 组合使用的 HTTP 被称为 HTTPS（HTTP Secure，超文本传输安全协议）或 HTTP over SSL。



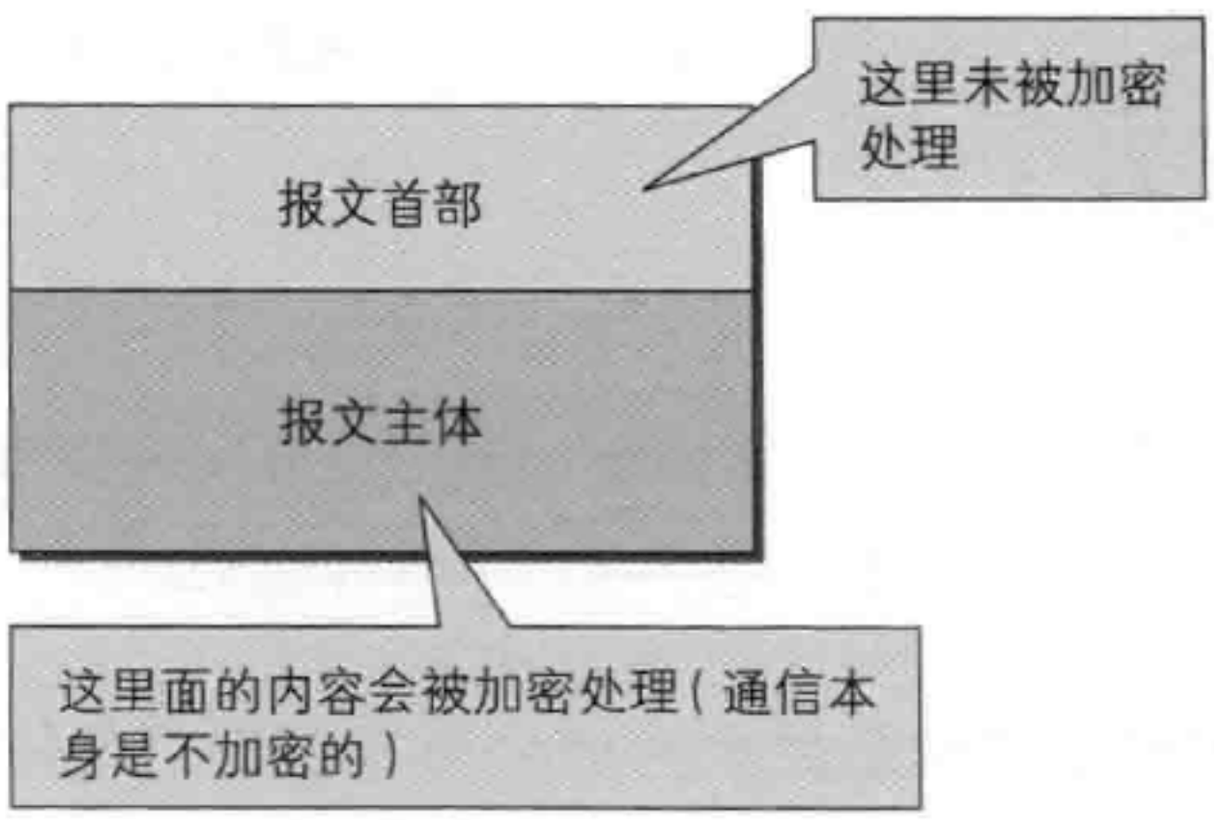
服务器与客户端之间建立起安全的通信线路之后开始通信



内容的加密

还有一种将参与通信的内容本身加密的方式。由于 HTTP 协议中没有加密机制，那么就对 HTTP 协议传输的内容本身加密。即把 HTTP 报文里所含的内容进行加密处理。

在这种情况下，客户端需要对 HTTP 报文进行加密处理后再发送请求。



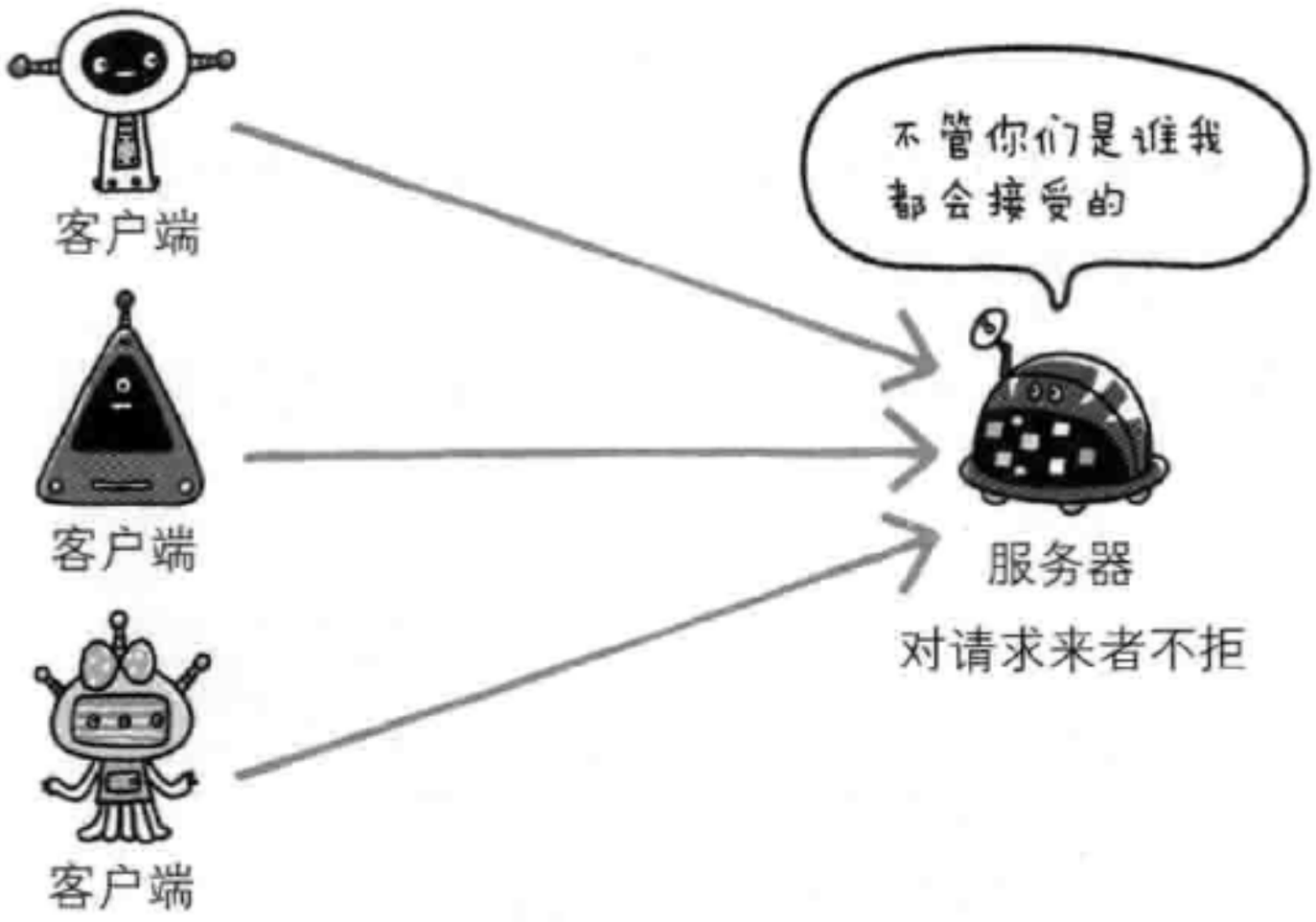
诚然，为了做到有效的内容加密，前提是要求客户端和服务器同时具备加密和解密机制。主要应用在 Web 服务中。有一点必须引起注意，由于该方式不同于 SSL 或 TLS 将整个通信线路加密处理，所以内容仍有被篡改的风险。稍后我们会加以说明。

7.1.2 不验证通信方的身份就可能遭遇伪装

HTTP 协议中的请求和响应不会对通信方进行确认。也就是说存在“服务器是否就是发送请求中 URI 真正指定的主机，返回的响应是否真的返回到实际提出请求的客户端”等类似问题。

■任何人都可发起请求

在 HTTP 协议通信时，由于不存在确认通信方的处理步骤，任何人都可以发起请求。另外，服务器只要接收到请求，不管对方是谁都会返回一个响应（但也仅限于发送端的 IP 地址和端口号没有被 Web 服务器设定限制访问的前提下）。



HTTP 协议的实现本身非常简单，不论是谁发送过来的请求都会返回响应，因此不确认通信方，会存在以下各种隐患。

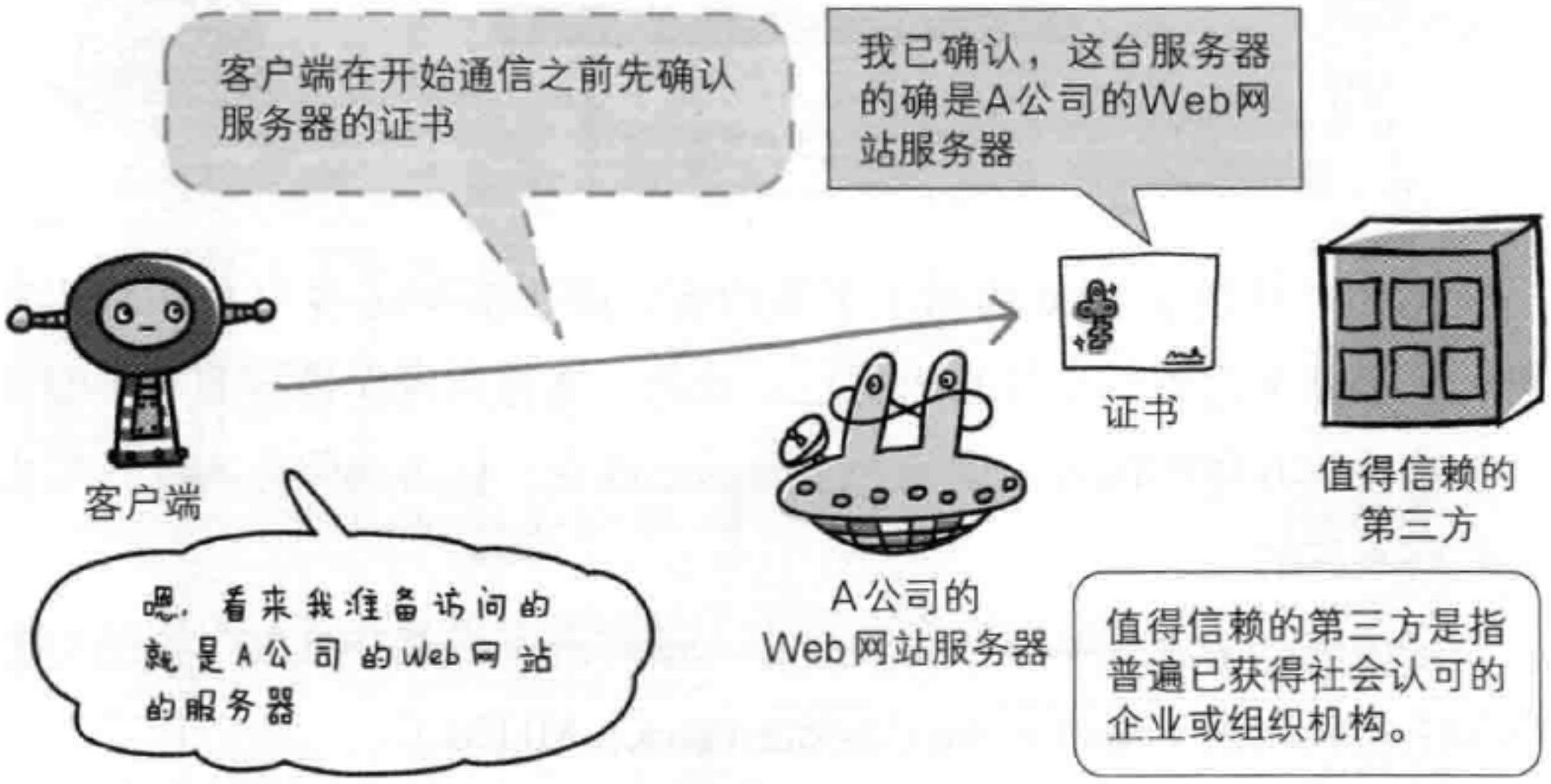
- 无法确定请求发送至目标的 Web 服务器是否是按真实意图返回响应的那台服务器。有可能是已伪装的 Web 服务器。
- 无法确定响应返回到的客户端是否是按真实意图接收响应的那个客户端。有可能是已伪装的客户端。
- 无法确定正在通信的对方是否具备访问权限。因为某些 Web 服务器上保存着重要的信息，只想发给特定用户通信的权限。

- 无法判定请求是来自何方、出自谁手。
- 即使是无意义的请求也会照单全收。无法阻止海量请求下的 DoS 攻击（Denial of Service，拒绝服务攻击）。

■查明对手的证书

虽然使用 HTTP 协议无法确定通信方，但如果使用 SSL 则可以。SSL 不仅提供加密处理，而且还使用了一种被称为证书的手段，可用于确定方。

证书由值得信任的第三方机构颁发，用以证明服务器和客户端是实际存在的。另外，伪造证书从技术角度来说是异常困难的一件事。所以只要能够确认通信方（服务器或客户端）持有的证书，即可判断通信方的真实意图。



通过使用证书，以证明通信方就是意料中的服务器。这对使用者个人来讲，也减少了个人信息泄露的危险性。

另外，客户端持有证书即可完成个人身份的确认，也可用于对 Web 网站的认证环节。

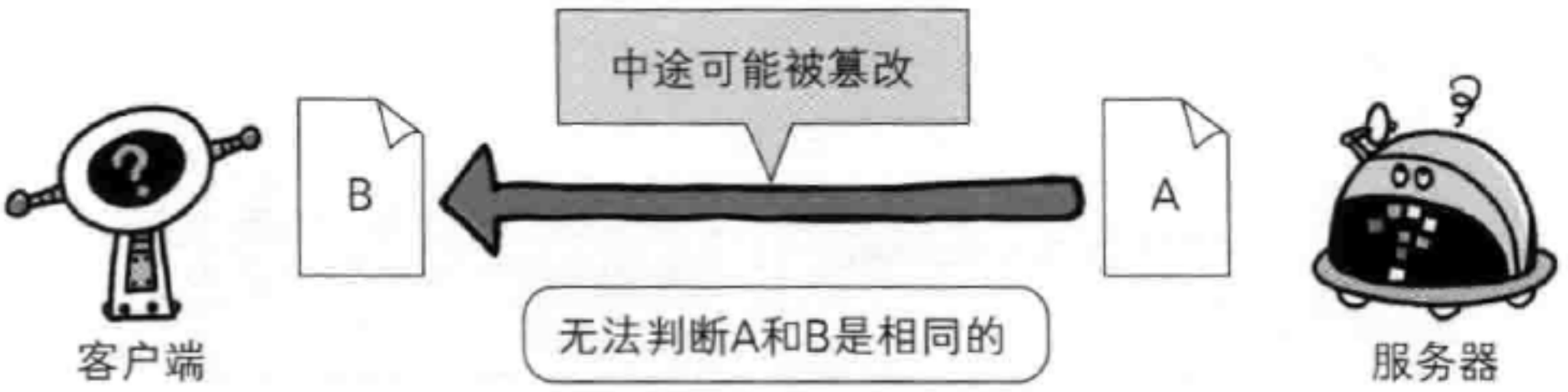
7.1.3 无法证明报文完整性，可能已遭篡改

所谓完整性是指信息的准确度。若无法证明其完整性，通常也就意味着无法判断信息是否准确。

■接收到的内容可能有误

由于 HTTP 协议无法证明通信的报文完整性，因此，在请求或响应送出之后直到对方接收之前的这段时间内，即使请求或响应的内容遭到篡改，也没有办法获悉。

换句话说，没有任何办法确认，发出的请求 / 响应和接收到的请求 / 响应是前后相同的。



比如，从某个 Web 网站上下载内容，是无法确定客户端下载的文件和服务器上存放的文件是否前后一致的。文件内容在传输途中可能已经被篡改为其他的内容。即使内容真的已改变，作为接收方的客户端也是觉察不到的。

像这样，请求或响应在传输途中，遭攻击者拦截并篡改内容的攻击称为中间人攻击（Man-in-the-Middle attack，MITM）。



图：中间人攻击

■如何防止篡改

虽然有使用 HTTP 协议确定报文完整性的方法，但事实上并不便捷、可靠。其中常用的是 MD5 和 SHA-1 等散列值校验的方法，以及用来确认文件的数字签名方法。

Apache HTTP Server 2.4.2 (httpd): 2.4.2 is the latest available version 2012-04-17

The Apache HTTP Server Project is pleased to announce the release of version 2.4.2 of the Apache HTTP Server ("Apache" and "httpd"). This version of Apache is our 2nd GA release of the new generation 2.4.x branch of Apache HTTPD and represents fifteen years of innovation by the project, and is recommended over all previous releases!

For details see the Official Announcement and the CHANGES_2.4 and CHANGES_2.4.2 lists

- Unix Source: httpd-2.4.2.tar.bz2 [PGP] [MD5] [SHA1]
- Unix Source: httpd-2.4.2.tar.gz [PGP] [MD5] [SHA1]
- Security and official patches
- Other files

Download - The Apache HTTP Server Project
http://httpd.apache.org/download.cgi

提供文件下载服务的 Web 网站也会提供相应的以 PGP（Pretty Good Privacy，完美隐私）创建的数字签名及 MD5 算法生成的散列值。PGP 是用来证明创建文件的数字签名，MD5 是由单向函数生成的散列值。

不论使用哪一种方法，都需要操纵客户端的用户本人亲自检查验证下载的文件是否就是原来服务器上的文件。浏览器无法自动帮用户检查。

可惜的是，用这些方法也依然无法百分百保证确认结果正确。因为PGP 和 MD5 本身被改写的话，用户是没有办法意识到的。

为了有效防止这些弊端，有必要使用 HTTPS。SSL 提供认证和加密处理及摘要功能。仅靠 HTTP 确保完整性是非常困难的，因此通过和其他协议组合使用来实现这个目标。下节我们介绍 HTTPS 的相关内容。

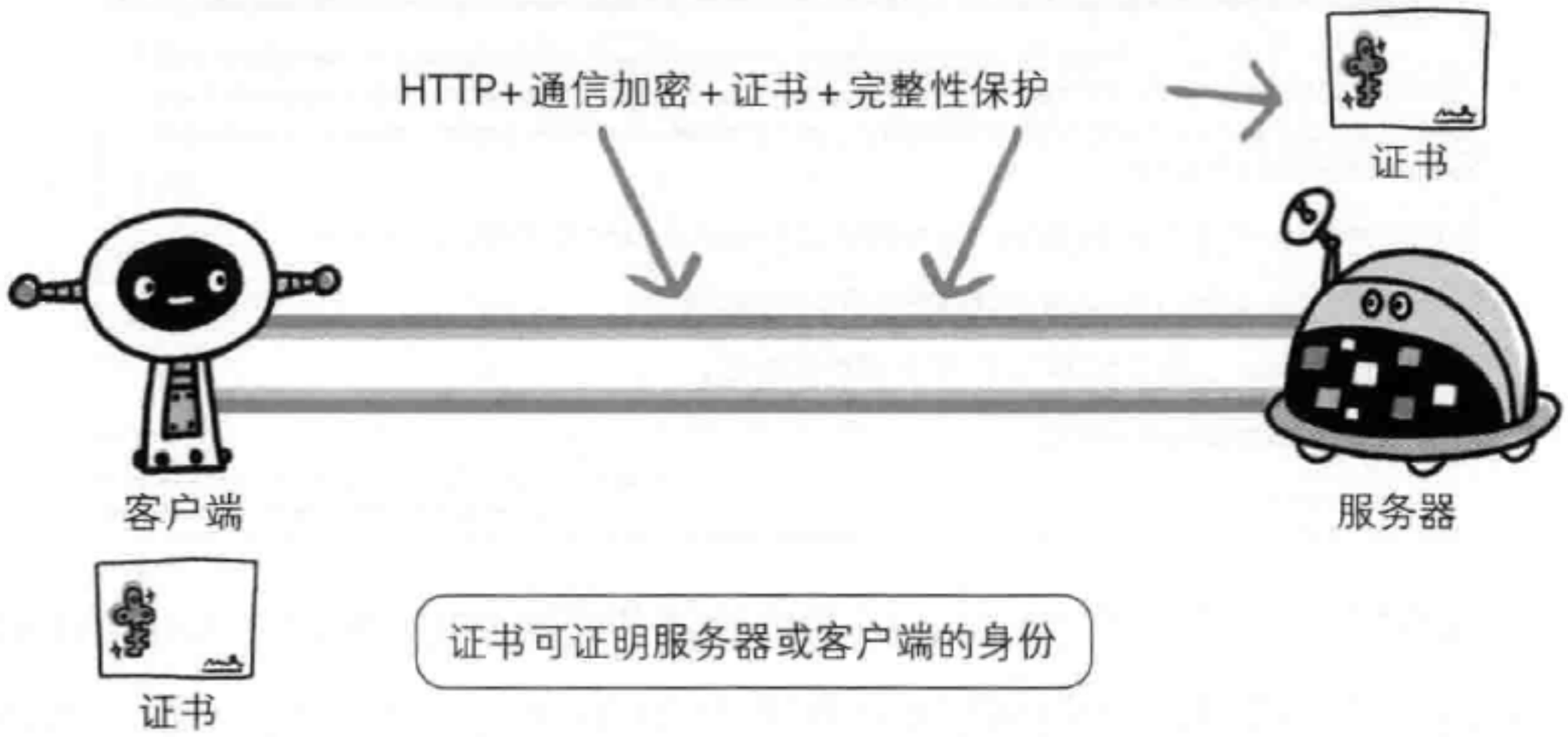
7.2 HTTP+ 加密 + 认证 + 完整性保护 =HTTPS

7.2.1 HTTP 加上加密处理和认证以及完整性保护后即是 HTTPS

如果在 HTTP 协议通信过程中使用未经加密的明文，比如在 Web 页面中输入信用卡号，如果这条通信线路遭到窃听，那么信用卡号就暴露了。

另外，对于 HTTP 来说，服务器也好，客户端也好，都是没有办法确认通信方的。因为很有可能并不是和原本预想的通信方在实际通信。并且还需要考虑到接收到的报文在通信途中已经遭到篡改这一可能性。

为了统一解决上述这些问题，需要在 HTTP 上再加入加密处理和认证等机制。我们把添加了加密及认证机制的 HTTP 称为 HTTPS（HTTP Secure）。



图：使用 HTTPS 通信

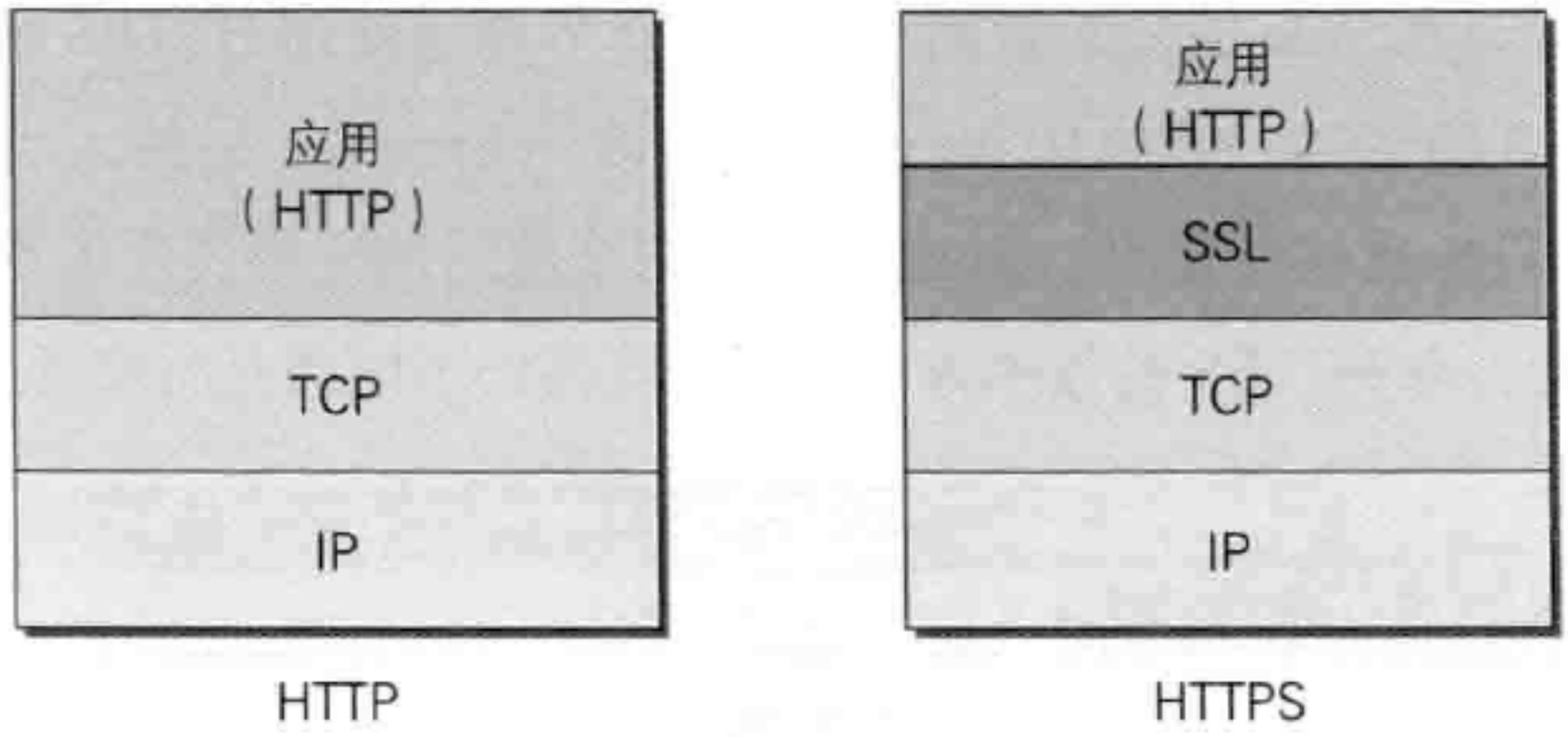
经常会在 Web 的登录页面和购物结算界面等使用 HTTPS 通信。使用 HTTPS 通信时，不再用 http://，而是改用 https://。另外，当浏览器访问 HTTPS 通信有效的 Web 网站时，浏览器的地址栏内会出现一个带锁的标记。对 HTTPS 的显示方式会因浏览器的不同而有所改变。



7.2.2　HTTPS 是身披 SSL 外壳的 HTTP

HTTPS 并非是应用层的一种新协议。只是 HTTP 通信接口部分用 SSL（Secure Socket Layer）和 TLS（Transport Layer Security）协议代替而已。

通常，HTTP 直接和 TCP 通信。当使用 SSL 时，则演变成先和 SSL 通信，再由 SSL 和 TCP 通信了。简言之，所谓 HTTPS，其实就是身披 SSL 协议这层外壳的 HTTP。



在采用 SSL 后，HTTP 就拥有了 HTTPS 的加密、证书和完整性保护这些功能。

SSL 是独立于 HTTP 的协议，所以不光是 HTTP 协议，其他运行在应用层的 SMTP 和 Telnet 等协议均可配合 SSL 协议使用。可以说 SSL 是当今世界上应用最为广泛的网络安全技术。

7.2.3 相互交换密钥的公开密钥加密技术

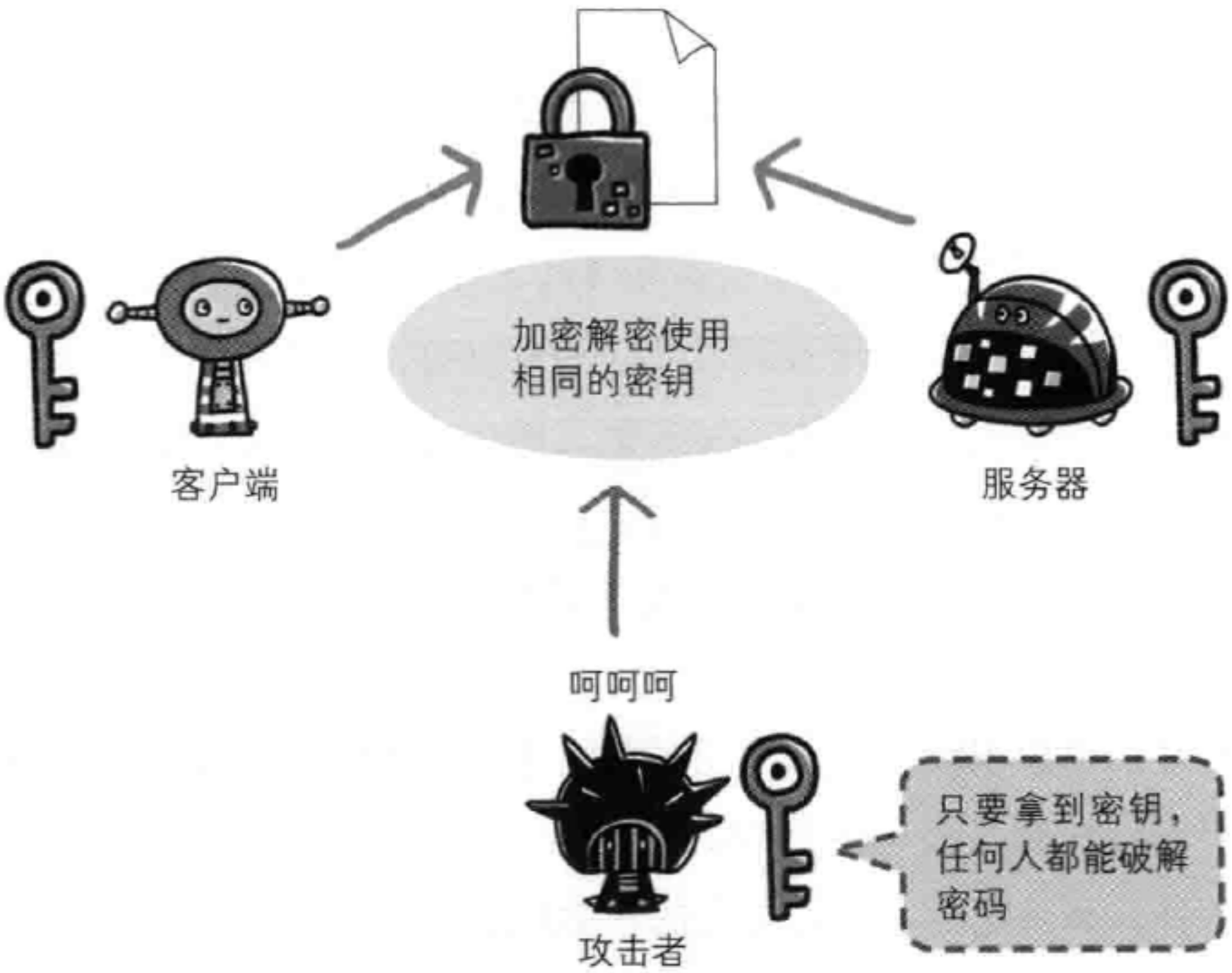
在对 SSL 进行讲解之前，我们先来了解一下加密方法。SSL 采用一种叫做公开密钥加密（Public-key cryptography）的加密处理方式。

近代的加密方法中加密算法是公开的，而密钥却是保密的。通过这种方式得以保持加密方法的安全性。

加密和解密都会用到密钥。没有密钥就无法对密码解密，反过来说，任何人只要持有密钥就能解密了。如果密钥被攻击者获得，那加密也就失去了意义。

■共享密钥加密的困境

加密和解密同用一个密钥的方式称为共享密钥加密（Common key crypto system），也被叫做对称密钥加密。



以共享密钥方式加密时必须将密钥也发给对方。可究竟怎样才能安全地转交？在互联网上转发密钥时，如果通信被监听那么密钥就可会落入攻击者之手，同时也就失去了加密的意义。另外还得设法安全地保管接收到的密钥。



发送密钥就有被窃听的风险，但不发送，对方就不能解密。再说，密钥若能够安全发送，那数据也应该能安全送达。

图：密钥发送问题

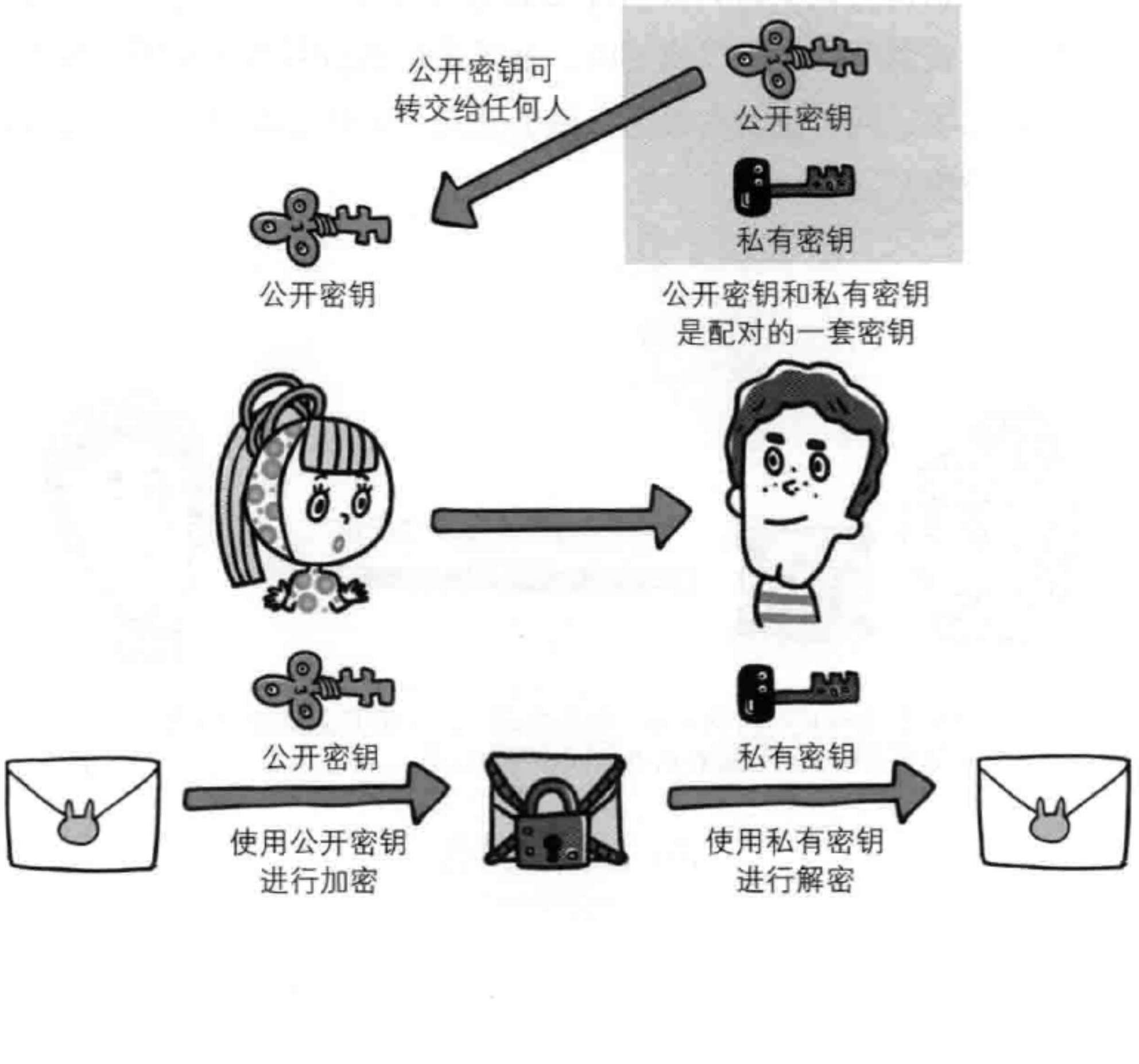
■使用两把密钥的公开密钥加密

公开密钥加密方式很好地解决了共享密钥加密的困难。

公开密钥加密使用一对非对称的密钥。一把叫做私有密钥（private key），另一把叫做公开密钥（public key）。顾名思义，私有密钥不能让其他任何人知道，而公开密钥则可以随意发布，任何人都可以获得。

使用公开密钥加密方式，发送密文的一方使用对方的公开密钥进行加密处理，对方收到被加密的信息后，再使用自己的私有密钥进行解密。利用这种方式，不需要发送用来解密的私有密钥，也不必担心密钥被攻击者窃听而盗走。

另外，要想根据密文和公开密钥，恢复到信息原文是异常困难的，因为解密过程就是在对离散对数进行求值，这并非轻而易举就能办到。退一步讲，如果能对一个非常大的整数做到快速地因式分解，那么密码破解还是存在希望的。但就目前的技术来看是不太现实的。



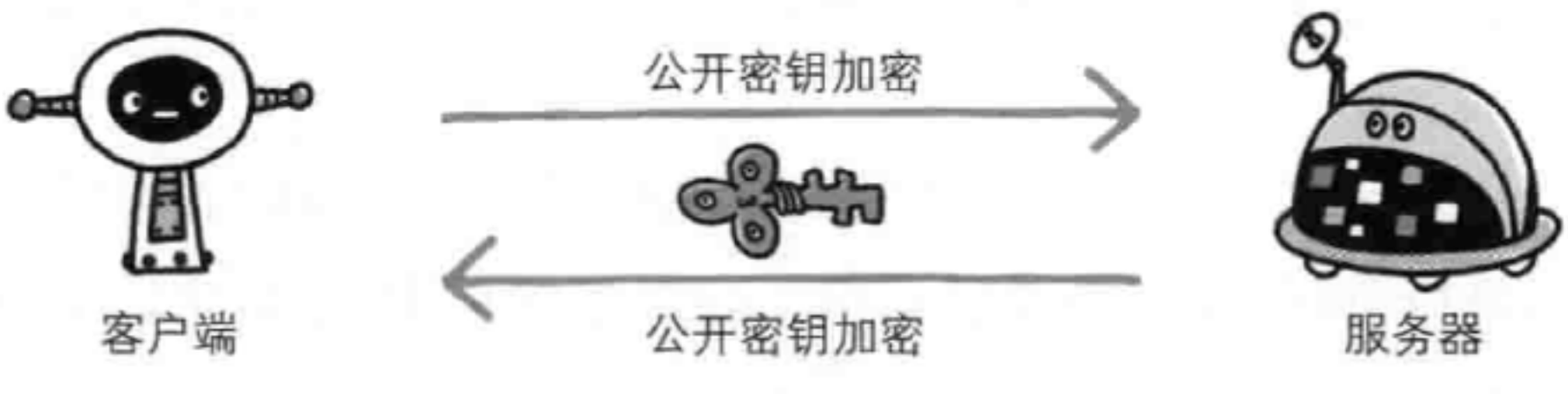
■HTTPS采用混合加密机制

HTTPS 采用共享密钥加密和公开密钥加密两者并用的混合加密机制。若密钥能够实现安全交换，那么有可能会考虑仅使用公开密钥加密来通信。但是公开密钥加密与共享密钥加密相比，其处理速度要慢。

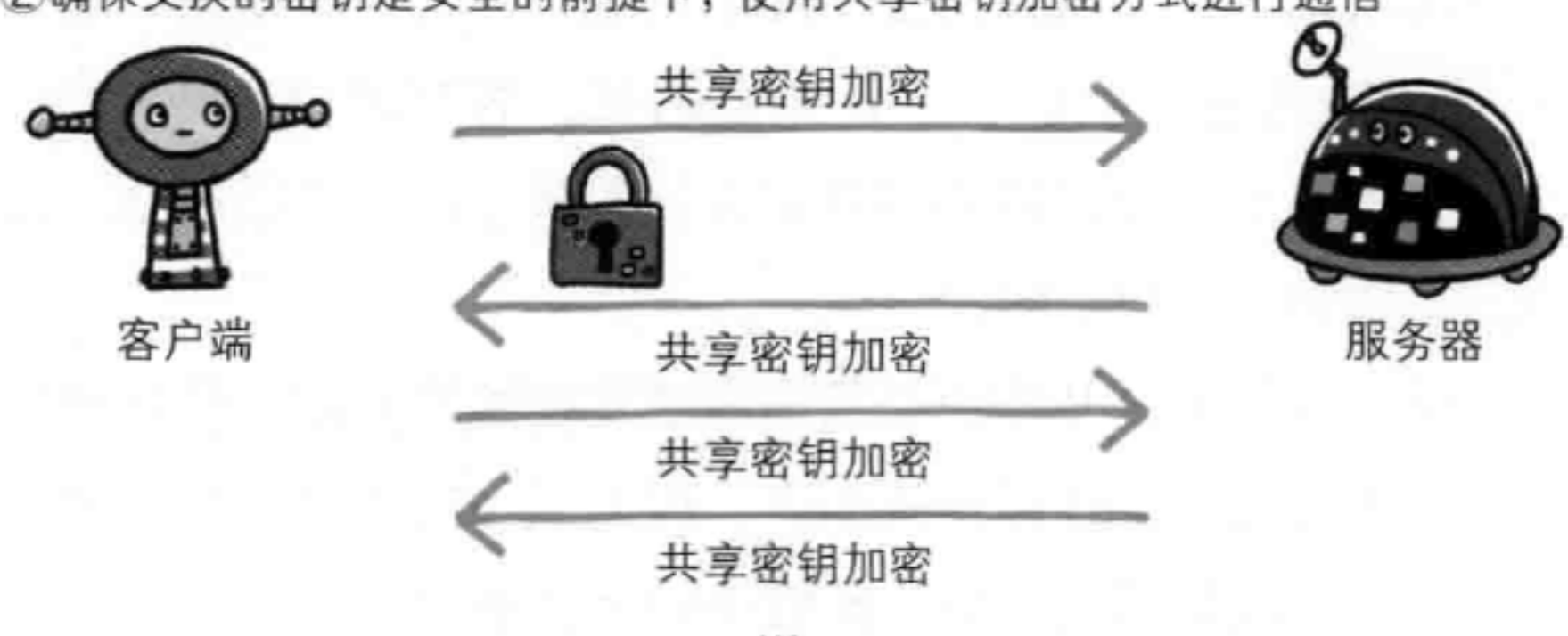
所以应充分利用两者各自的优势，将多种方法组合起来用于通信。在交换密钥环节使用公开密钥加密方式，之后的建立通信交换报文阶段则使用共享密钥加密方式。

> 公开密钥加密处理起来比共享密钥加密方式更为复杂，因此若在通信时使用公开密钥加密方式，效率就很低

①使用公开密钥加密方式安全地交换在稍后的共享密钥加密中要使用的密钥



②确保交换的密钥是安全的前提下，使用共享密钥加密方式进行通信



…

图：混合加密机制

7.2.4　证明公开密钥正确性的证书

遗憾的是，公开密钥加密方式还是存在一些问题的。那就是无法证明公开密钥本身就是货真价实的公开密钥。比如，正准备和某台服务器

建立公开密钥加密方式下的通信时，如何证明收到的公开密钥就是原本预想的那台服务器发行的公开密钥。或许在公开密钥传输途中，真正的公开密钥已经被攻击者替换掉了。

为了解决上述问题，可以使用由数字证书认证机构（CA，Certificate Authority）和其相关机关颁发的公开密钥证书。

数字证书认证机构处于客户端与服务器双方都可信赖的第三方机构的立场上。威瑞信（VeriSign）就是其中一家非常有名的数字证书认证机构。我们来介绍一下数字证书认证机构的业务流程。首先，服务器的运营人员向数字证书认证机构提出公开密钥的申请。数字证书认证机构在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名，然后分配这个已签名的公开密钥，并将该公开密钥放入公钥证书后绑定在一起。

服务器会将这份由数字证书认证机构颁发的公钥证书发送给客户端，以进行公开密钥加密方式通信。公钥证书也可叫做数字证书或直接称为证书。

接到证书的客户端可使用数字证书认证机构的公开密钥，对那张证书上的数字签名进行验证，一旦验证通过，客户端便可明确两件事：一，认证服务器的公开密钥的是真实有效的数字证书认证机构。二，服务器的公开密钥是值得信赖的。

此处认证机关的公开密钥必须安全地转交给客户端。使用通信方式时，如何安全转交是一件很困难的事，因此，多数浏览器开发商发布版本时，会事先在内部植入常用认证机关的公开密钥。



数字证书认证机构的公开密钥已事先植入到浏览器里了



数字证书认证机构



数字证书认证机构的私有密钥

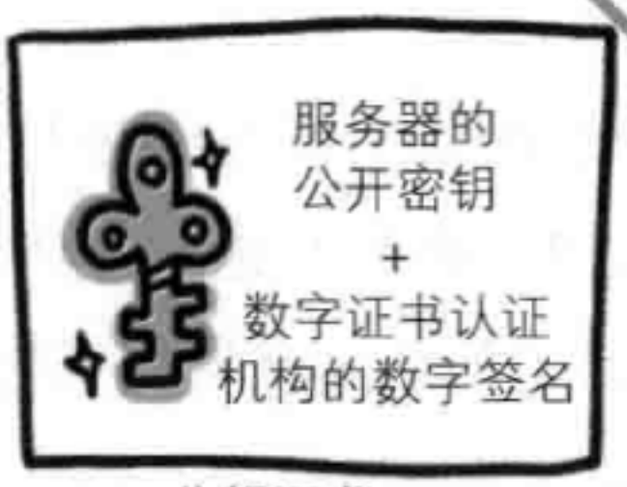
②数字证书认证机构用自己的私有密钥向服务器的公开密码署数字签名并颁发公钥证书

①服务器把自己的公开密钥登录至数字证书认证机构



服务器的公开密钥

③客户端拿到服务器的公钥证书后，使用数字证书认证机构的公开密钥，向数字证书认证机构验证公钥证书上的数字签名，以确认服务器的公开密钥的真实性



公钥证书



客户端

④使用服务器的公开密钥对报文加密后发送

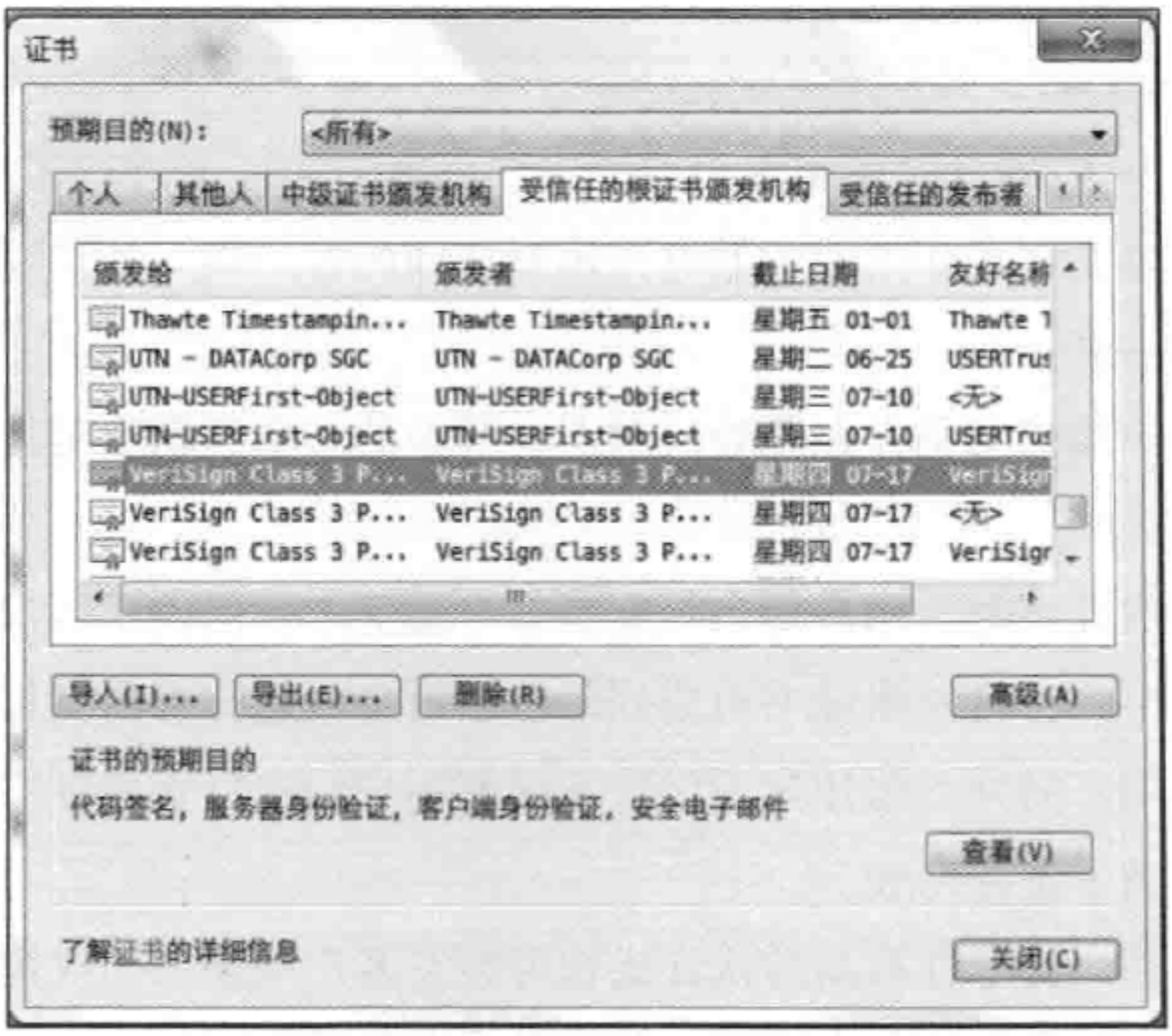


服务器



服务器的私有密钥

⑤服务器用私有密钥对报文解密



■可证明组织真实性的EV SSL证书

证书的一个作用是用来证明作为通信一方的服务器是否规范，另外一个作用是可确认对方服务器背后运营的企业是否真实存在。拥有该特性的证书就是 EV SSL 证书（Extended Validation SSL Certificate）。

EV SSL 证书是基于国际标准的认证指导方针颁发的证书。其严格规定了对运营组织是否真实的确认方针，因此，通过认证的 Web 网站能够获得更高的认可度。

持有 EV SSL 证书的 Web 网站的浏览器地址栏处的背景色是绿色的，从视觉上就能一眼辨别出。而且在地址栏的左侧显示了 SSL 证书中记录的组织名称以及颁发证书的认证机构的名称。



上述机制的原意图是为了防止用户被钓鱼攻击（Phishing），但就效果上来讲，还得打一个问号。很多用户可能不了解 EV SSL 证书相关的知识，因此也不太会留意它。

■用以确认客户端的客户端证书

HTTPS 中还可以使用客户端证书。以客户端证书进行客户端认证，证明服务器正在通信的对方始终是预料之内的客户端，其作用跟服务器证书如出一辙。

但客户端证书仍存在几处问题点。其中的一个问题点是证书的获取及发布。

想获取证书时，用户得自行安装客户端证书。但由于客户端证书是要付费购买的，且每张证书对应到每位用户也就意味着需支付和用户数对等的费用。另外，要让知识层次不同的用户们自行安装证书，这件事本身也充满了各种挑战。

现状是，安全性极高的认证机构可颁发客户端证书但仅用于特殊用途的业务。比如那些可支撑客户端证书支出费用的业务。

例如，银行的网上银行就采用了客户端证书。在登录网银时不仅要求用户确认输入 ID 和密码，还会要求用户的客户端证书，以确认用户是否从特定的终端访问网银。

客户端证书存在的另一个问题点是，客户端证书毕竟只能用来证明客户端实际存在，而不能用来证明用户本人的真实有效性。也就是说，只要获得了安装有客户端证书的计算机的使用权限，也就意味着同时拥有了客户端证书的使用权限。

■认证机构信誉第一

SSL 机制中介入认证机构之所以可行，是因为建立在其信用绝对可靠这一大前提下的。然而，2011 年 7 月，荷兰的一家名叫 DigiNotar 的认证机构曾遭黑客不法入侵，颁布了 google.com 和 twitter.com 等网站的伪造证书事件。这一事件从根本上撼动了 SSL 的可信度。

因为伪造证书上有正规认证机构的数字签名，所以浏览器会判定该证书是正当的。当伪造的证书被用做服务器伪装之时，用户根本无法察觉到。

虽然存在可将证书无效化的证书吊销列表（Certificate Revocation List，CRL）机制，以及从客户端删除根证书颁发机构（Root Certificate Authority，RCA）的对策，但是距离生效还需要一段时间，而在这段时间内，到底会有多少用户的利益蒙受损失就不得而知了。

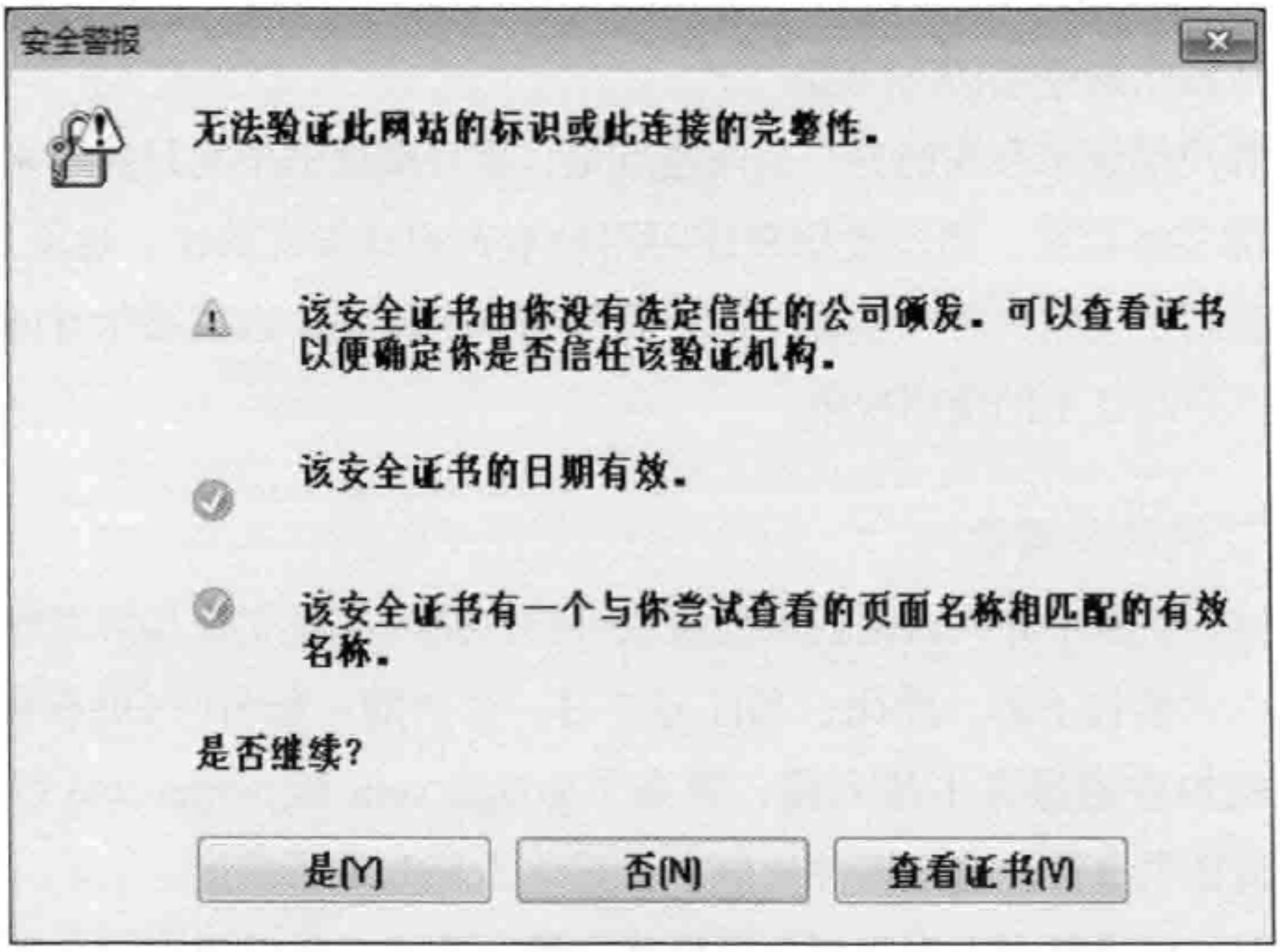
■由自认证机构颁发的证书称为自签名证书

如果使用 OpenSSL 这套开源程序，每个人都可以构建一套属于自己的认证机构，从而自己给自己颁发服务器证书。但该服务器证书在互联网上不可作为证书使用，似乎没什么帮助。

独立构建的认证机构叫做自认证机构，由自认证机构颁发的“无用”证书也被戏称为自签名证书。

浏览器访问该服务器时，会显示“无法确认连接安全性”或“该网

站的安全证书存在问题”等警告消息。



由自认证机构颁发的服务器证书之所以不起作用，是因为它无法消除伪装的可能性。自认证机构能够产生的作用顶多也就是自己对外宣称“我是○○”的这种程度。即使采用自签名证书，通过 SSL 加密之后，可能偶尔还会看见通信处在安全状态的提示，可那也是有问题的。因为就算加密通信，也不能排除正在和已经过伪装的假服务器保持通信。

值得信赖的第三方机构介入认证，才能让已植入在浏览器内的认证机构颁布的公开密钥发挥作用，并借此证明服务器的真实性。

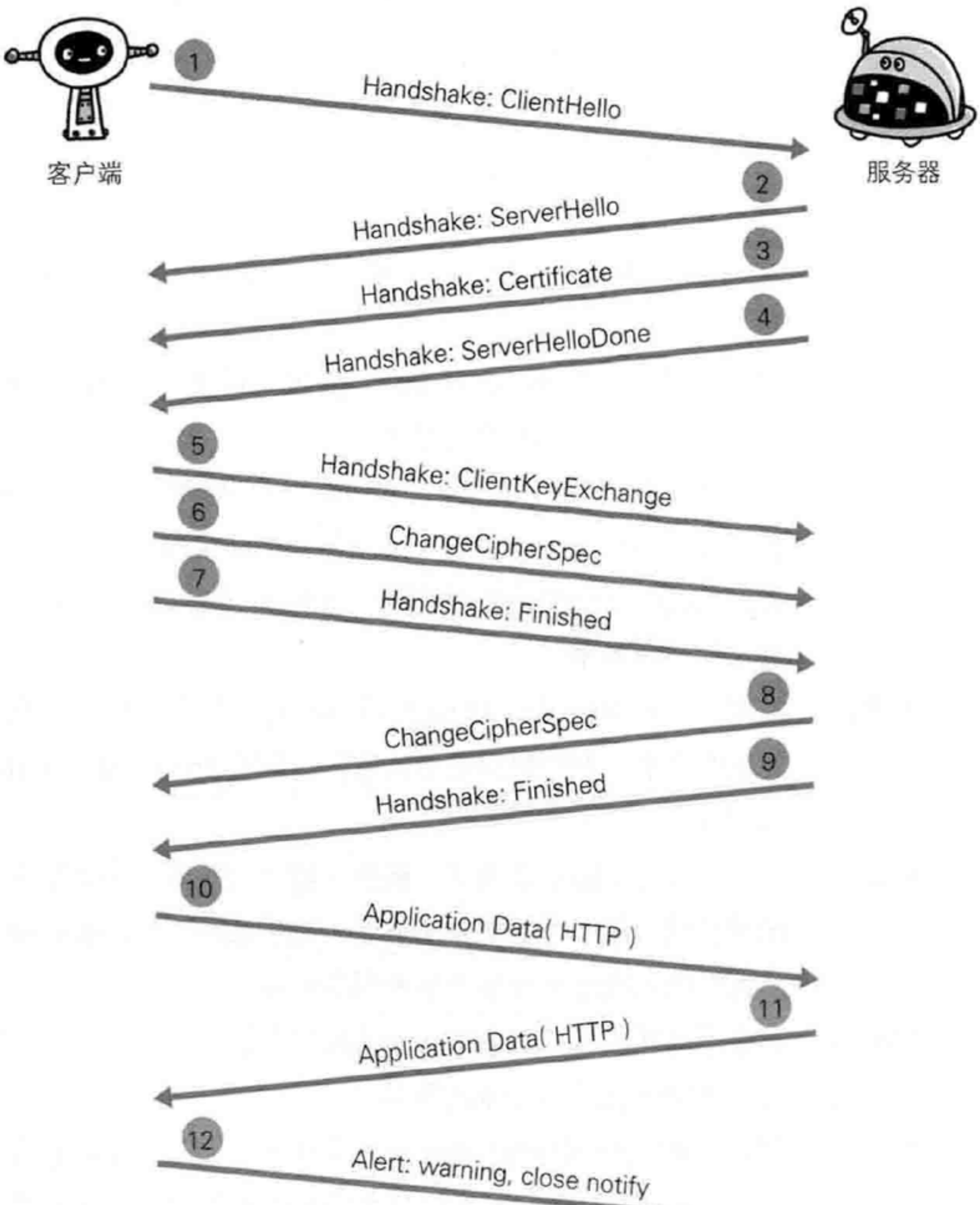
中级认证机构的证书可能会变成自认证证书

多数浏览器内预先已植入备受信赖的认证机构的证书，但也有一小部分浏览器会植入中级认证机构的证书。

对于中级认证机构颁发的服务器证书，某些浏览器会以正规的证书来对待，可有的浏览器会当作自签名证书。

7.2.5　HTTPS 的安全通信机制

为了更好地理解 HTTPS，我们来观察一下 HTTPS 的通信步骤。



图：HTTPS 通信

步骤 1：客户端通过发送 Client Hello 报文开始 SSL 通信。报文中包含客户端支持的 SSL 的指定版本、加密组件（Cipher Suite）列表（所使用的加密算法及密钥长度等）。

步骤 2：服务器可进行 SSL 通信时，会以 Server Hello 报文作为应答。和客户端一样，在报文中包含 SSL 版本以及加密组件。服务器的加密组件内容是从接收到的客户端加密组件内筛选出来的。

步骤 3：之后服务器发送 Certificate 报文。报文中包含公开密钥证书。

步骤 4：最后服务器发送 Server Hello Done 报文通知客户端，最初阶段的 SSL 握手协商部分结束。

步骤 5：SSL 第一次握手结束之后，客户端以 Client Key Exchange 报文作为回应。报文中包含通信加密中使用的一种被称为 Pre-master secret 的随机密码串。该报文已用步骤 3 中的公开密钥进行加密。

步骤 6：接着客户端继续发送 Change Cipher Spec 报文。该报文会提示服务器，在此报文之后的通信会采用 Pre-master secret 密钥加密。

步骤 7：客户端发送 Finished 报文。该报文包含连接至今全部报文的整体校验值。这次握手协商是否能够成功，要以服务器是否能够正确解密该报文作为判定标准。

步骤 8：服务器同样发送 Change Cipher Spec 报文。

步骤 9：服务器同样发送 Finished 报文。

步骤 10：服务器和客户端的 Finished 报文交换完毕之后，SSL 连接就算建立完成。当然，通信会受到 SSL 的保护。从此处开始进行应用层协议的通信，即发送 HTTP 请求。

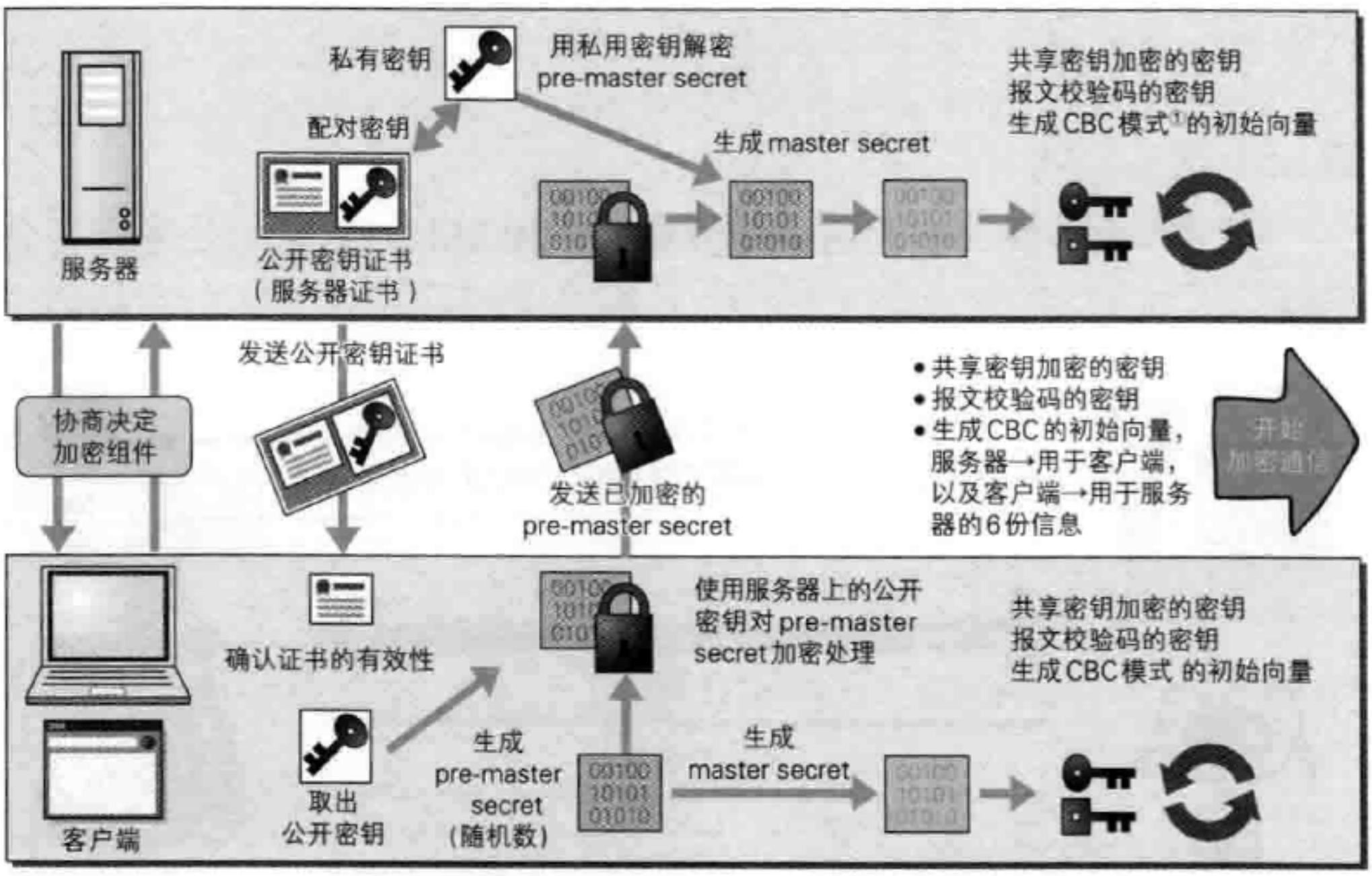
步骤 11：应用层协议通信，即发送 HTTP 响应。

步骤 12：最后由客户端断开连接。断开连接时，发送 close_notify

报文。上图做了一些省略，这步之后再发送 TCP FIN 报文来关闭与 TCP 的通信。

在以上流程中，应用层发送数据时会附加一种叫做 MAC（Message Authentication Code）的报文摘要。MAC 能够查知报文是否遭到篡改，从而保护报文的完整性。

下面是对整个流程的图解。图中说明了从仅使用服务器端的公开密钥证书（服务器证书）建立 HTTPS 通信的整个过程。



SSL和TLS

HTTPS 使用 SSL（Secure Socket Layer）和 TLS（Transport Layer Security）这两个协议。

① CBC模式（Cipher Block Chaining）又名密码分组链接模式。在此模式下，将前一个明文块加密处理后和下一个明文块做XOR运算，使之重叠，然后再对运算结果做加密处理。对第一个明文块做加密时，要么使用前一段密文的最后一块，要么利用外部生成的初始向量（initial vector，IV）。——译者注

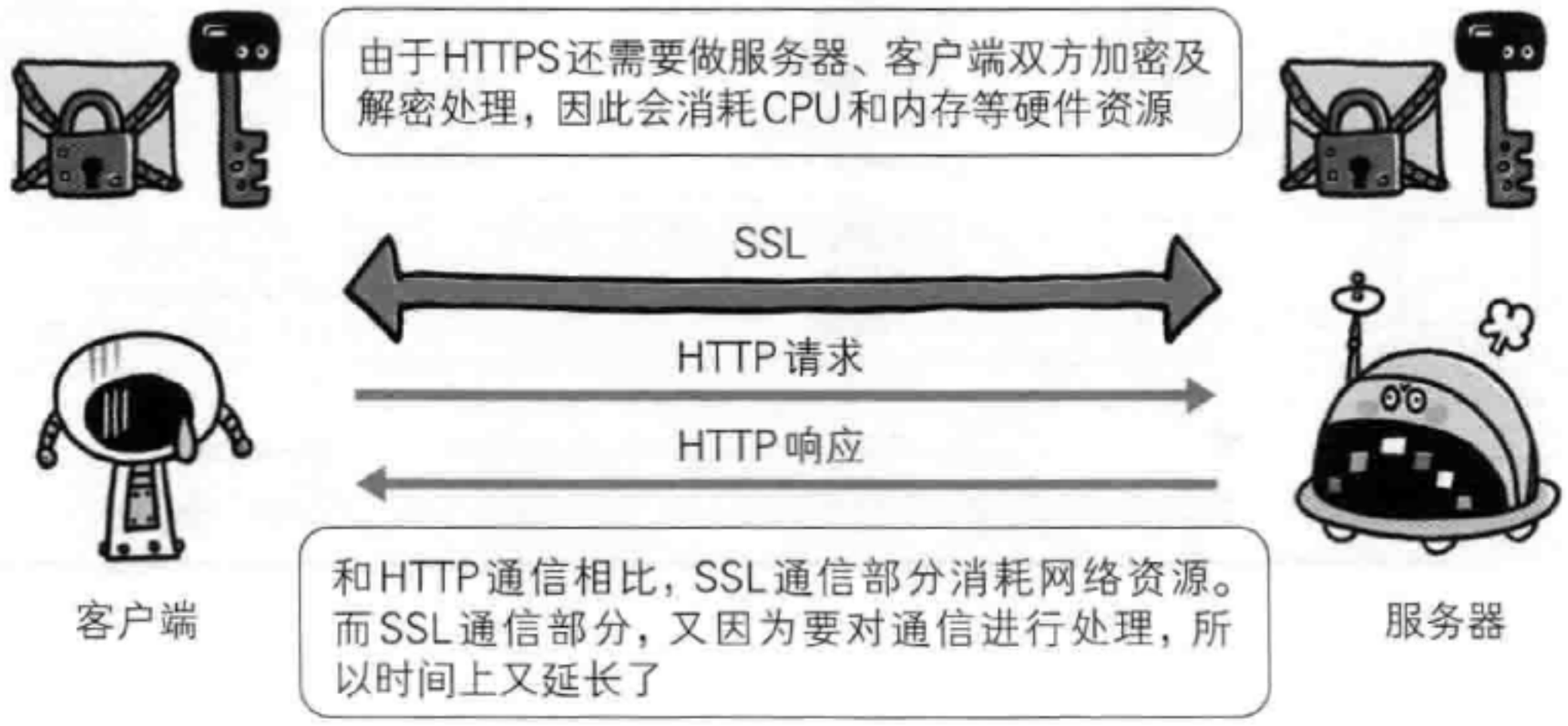
SSL 技术最初是由浏览器开发商网景通信公司率先倡导的，开发过 SSL3.0 之前的版本。目前主导权已转移到 IETF（Internet Engineering Task Force，Internet 工程任务组）的手中。

IETF 以 SSL3.0 为基准，后又制定了 TLS1.0、TLS1.1 和 TLS1.2。TSL 是以 SSL 为原型开发的协议，有时会统一称该协议为 SSL。当前主流的版本是 SSL3.0 和 TLS1.0。

由于 SSL1.0 协议在设计之初被发现出了问题，就没有实际投入使用。SSL2.0 也被发现存在问题，所以很多浏览器直接废除了该协议版本。

■SSL 速度慢吗

HTTPS 也存在一些问题，那就是当使用 SSL 时，它的处理速度会变慢。



图：HTTPS 比 HTTP 要慢 2 到 100 倍

SSL 的慢分两种。一种是指通信慢。另一种是指由于大量消耗 CPU 及内存等资源，导致处理速度变慢。

和使用 HTTP 相比，网络负载可能会变慢 2 到 100 倍。除去和 TCP 连接、发送 HTTP 请求·响应以外，还必须进行 SSL 通信，因此整体上处理通信量不可避免会增加。

另一点是 SSL 必须进行加密处理。在服务器和客户端都需要进行加密和解密的运算处理。因此从结果上讲，比起 HTTP 会更多地消耗服务器和客户端的硬件资源，导致负载增强。

针对速度变慢这一问题，并没有根本性的解决方案，我们会使用 SSL 加速器这种（专用服务器）硬件来改善该问题。该硬件为 SSL 通信专用硬件，相对软件来讲，能够提高数倍 SSL 的计算速度。仅在 SSL 处理时发挥 SSL 加速器的功效，以分担负载。

为什么不一直使用 HTTPS

既然 HTTPS 那么安全可靠，那为何所有的 Web 网站不一直使用 HTTPS？

其中一个原因是，因为与纯文本通信相比，加密通信会消耗更多的 CPU 及内存资源。如果每次通信都加密，会消耗相当多的资源，平摊到一台计算机上时，能够处理的请求数量必定也会随之减少。

因此，如果是非敏感信息则使用 HTTP 通信，只有在包含个人信息等敏感数据时，才利用 HTTPS 加密通信。

特别是每当那些访问量较多的 Web 网站在进行加密处理时，它们所承担着的负载不容小觑。在进行加密处理时，并非对所有内容都进行加密处理，而是仅在那些需要信息隐藏时才会加密，以节约资源。



除此之外，想要节约购买证书的开销也是原因之一。