



Unicode et les conséquences sur PHP 6

En novembre 2005, les développeurs de PHP se rencontrèrent à Paris pour discuter des évolutions de la sixième version du langage. Cinq ans plus tard, PHP 6 n'est toujours pas disponible. Examinons ensemble les causes de ce retard.

Les premières annonces de PHP 6 étaient très prometteuses :

- Ajout des namespaces : possibilité de lier des classes ou des fonctions à un « espace de nom ». Ces classes seront disponibles uniquement si on importe explicitement le namespace (voir le mot-clé « import » en Java ou Python) ;
- Static binding : permet d'effectuer la résolution des attributs statiques par la classe appelante et non par la classe mère ;
- Ajout des fonctions anonymes ;
- Du nettoyage : suppression des « register globals », « magic quotes » et « safe mode » ;
- Intégration d'un « goto » ;
- Intégration d'un cache de bytecode ;
- Et surtout l'Unicode.

Unicode en PHP.

L'ambition de PHP est d'intégrer l'Unicode au cœur du langage. Ceci doit permettre l'utilisation de n'importe quel caractère dans le code source (oui, vous pourrez mettre des accents dans vos noms de variables, mais ne vous sentez pas obligé de le faire !). Mais le support de l'Unicode devrait surtout permettre d'utiliser des sources de données qui contiennent autre chose que de l'ASCII, de manière native, sans utiliser de bibliothèque externe (adieu mbstring). Les caractères de la langue anglaise ne seraient qu'un sous-ensemble du jeu de caractères supporté, et non pas une langue privilégiée. Voici un exemple qui met en évidence le fait que PHP ne gère pas l'Unicode. Prenons un fichier XML dont l'encoding est définie explicitement comme UTF-8.

```
$ cat exemple.xml
<?xml version="1.0" encoding="UTF-8"?>
<example>
  <string>_____ unicode</string>
</example>
```

Voici un parseur XML trivial (il n'y a aucune vérification d'erreurs). Le parseur utilise SimpleXML.

```
<?php
$xml = simplexml_load_file("exemple.xml");
$results = $xml->xpath('/example/string');
$str = $results[0];

echo "R_sultat simpleXML : ".$str;
echo "ÃnÃn";

echo "API de base:Ãn";
echo "taille de la chaîne : ".strlen($str);
echo "Ãn";
```

```
for($i=0; $i < strlen($str); $i++) {
    echo substr($str, $i, 1);
    echo " ";
}

echo "ÃnÃnAPI unicode:Ãn";
echo "taille de la chaîne : ".mb_strlen($str, "utf-8");
echo "Ãn";

for($i=0; $i < mb_strlen($str, "utf-8"); $i++) {
    echo mb_substr($str, $i, 1, "utf-8");
    echo " ";
}

echo "Ãn";
```

Et la sortie :

```
R_sultat simpleXML : _____ unicode

API de base:
taille de la chaîne : 21
_____ unicode

API unicode:
taille de la chaîne : 14
_____ unicode
```

On voit que les fonctions de base de PHP considèrent que tous les caractères sont codés sur 1 octet. Pour utiliser du texte Unicode, il faut passer par l'API « multi-byte », avec les fonctions mbstring (on retrouve les mêmes noms que pour l'API « strings » mais avec le préfixe « mb_ »). Pour le développeur, la prise en charge d'Unicode devrait faciliter la manipulation des bases de données ou des fichiers XML depuis un script PHP. Pour l'utilisateur, on peut enfin espérer la fin du phénomène « Mojibake », ces fameux bugs d'affichage dans les pages web ou les e-mails. Qui n'a jamais vu des chaînes de caractères tels que :

Bonjour C_`dric

Quelques mots sur l'Unicode.

Le but d'Unicode est d'attribuer un nom et un identifiant numérique à tous les caractères, quelle que soit la langue. On compte aujourd'hui environ 245 000 points (signes) définis. La norme Unicode répartit les points dans 17 différents plans :

- Le plus utilisé est le « plan multilingue de base » (PMB). Il contient les points 0x0000 à 0xFFFF (il faut donc 2 octets pour le coder). Ce plan contient les caractères les plus utilisées dans la majorité des textes, c'est-à-dire, entre autres, les alphabets latin, cyril-



lique, grec, arabe, hébreu, ainsi qu'une partie des signes utilisés par le chinois, japonais et coréen.

- Le « plan multilingue complémentaire » (PMC). Il contient les points 0x10000 à 0x1FFFF. Ce plan permet de représenter les caractères rares, comme le gothique, les idéogrammes du linéaire B ou des symboles mathématiques.
- Le « Plan idéographique complémentaire » (PIC) (0x20000 - 0x2FFFF). Il contient des idéogrammes complémentaires pour le chinois, japonais et coréen.

Les autres plans sont soit réservés, soit à usage privé.

L'espace d'Unicode peut contenir un peu plus d'un million de points, codés sur 4 octets.

Il faut comprendre qu'un point ne correspond pas directement à un caractère. Par exemple le glyphe « é » peut être décrit soit directement par son numéro de point, soit par celui du « e » suivi du point « accent aigu ». Ces deux représentations sont strictement équivalentes et doivent être traitées de la même façon. A contrario, certains signes comme la ligature « fi » peuvent être représentés par un seul point Unicode.

Une fois que l'on connaît les différents points Unicode de notre texte, il faut les coder afin de pouvoir les échanger ou en faire une représentation physique en mémoire.

Nous allons présenter les 3 principaux encodages utilisés.

UTF-8

C'est un codage à taille variable. Tous les caractères de la table ASCII sont présents dans UTF-8 sans changement. Un texte ASCII est donc déjà converti en UTF-8 (donc sur 1 octet).

Ceci est très important pour la compatibilité avec l'existant et l'adoption de l'Unicode. C'est aussi ce qui explique pourquoi cet encodage est le plus connu aujourd'hui.

Les caractères latin, grec, cyrillique, arabe, hébreu sont codés sur 2 octets. Le reste du plan PMB nécessite au plus 3 octets. Tous les autres caractères nécessitent 4 octets. Un autre avantage d'UTF-8 est qu'il est indépendant de l'endianness de la machine. Par contre, le fait d'être un codage à taille variable complexifie les opérations sur les chaînes de caractères (il faut parcourir la chaîne pour connaître le caractère à la position N). UTF-8 est souvent utilisé pour l'échange de données.

UTF-16

Tous les caractères du plan PMB sont codés sur 2 octets. Pour les autres caractères il faut utiliser 2 paires d'octets. Pour l'échange d'information, il est nécessaire d'ajouter une marque pour indiquer l'ordonnement des octets de la machine (BOM, Byte Order Mark). UTF-16 est souvent utilisé pour les traitements internes à une application.

UTF-32

Les identifiants Unicode sont directement codés sur 4 octets. Comme pour UTF-16, un BOM est nécessaire. Cette représentation coûteuse en mémoire n'est utilisée que pour des traitements internes où la rapidité de traitement est importante.

L'approche choisie par PHP.

Pour intégrer l'Unicode, les développeurs de PHP ont décidé de s'appuyer sur une bibliothèque déjà existante : International Components for Unicode (ICU). Il se trouve que ICU utilise UTF-16 en



interne, c'est donc naturellement que PHP a aussi choisi UTF-16 pour le codage de toutes les chaînes de caractères. L'idée est de convertir les données une fois en entrée, depuis le système de codage de l'utilisateur vers UTF-16 et de les re-transformer en sortie. Toutes les manipulations internes se font en UTF-16.

Le problème est que les conversions s'avèrent être nombreuses et coûteuses. Dans le monde du web, les données sont souvent en UTF-8 (base de données, fichier XML, ...). Convertir une chaîne depuis UTF-8 vers UTF-16 coûte cher en CPU et en mémoire (dans la plupart des cas, la chaîne UTF-16 sera deux fois plus grande qu'en UTF-8). Ceci complexifie l'implémentation et ajoute des problèmes de compatibilité avec le code existant.

Tout ceci fait que de nombreux développeurs s'éloignent de PHP 6. Certains préféreraient adopter une approche plus pragmatique pour résoudre le problème de l'Unicode. Par exemple, une meilleure intégration des fonctions « mbstring », sans changer fondamentalement le moteur. D'autres encore utilisent la branche PHP 5 pour proposer des évolutions du langage, laissant les développements sur PHP 6 stagner.

Le renouveau

L'année dernière, la branche 5.3 fut créée afin de prendre des éléments du futur PHP 6 et les intégrer dans PHP 5.

En mars 2010, Rasmus Lerdorf décida de déplacer le code qui devait constituer la future version PHP 6 vers une branche et de reprendre le développement principal à partir de la série 5.3. Les évolutions intéressantes pourront toujours être reprises depuis l'ancienne branche PHP 6, qui sera maintenant dédiée à la résolu-

tion du « problème Unicode ». Le but de cette réorganisation est de relancer le développement de PHP qui souffrait d'un manque d'enthousiasme. La communauté PHP est toujours active et va pouvoir recentrer ses efforts sur les évolutions à apporter au langage. Pour ceux qui se demandent à quelle date va sortir PHP 6, cette question n'est pas (n'est plus, N.D.L.R.) à l'ordre du jour. Lorsque suffisamment de fonctionnalités seront développées, une nouvelle version du langage sortira, mais on ne sait pas si ce sera un PHP 5.4, un PHP 6 ou même directement un PHP 7. ... À moins que Perl 6 ne sorte avant :-)

Ressources

Resetting PHP 6
<http://www.net/articles/379909/>

[PHP-DEV] PHP 6
<http://thread.gmane.org/gmane.comp.php.devel/60062>

Minutes PHP Developers Meeting
<http://www.php.net/~derick/meeting-notes.html>

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
<http://www.joelonsoftware.com/articles/Unicode.html>

Unicode
<http://fr.wikipedia.org/wiki/Unicode>

■ **Cédric Cabessa**
*Ingénieur développement opensource.
Groupe Devoteam, IDS-Uperto*