PHP8

تكليف تطوير الويب متقدم عملي

الدكتور

إبراهيم الشامي

سمر عزي محمد الكرعمي

تقنية معلومات

مستوى ثالث

ما الفرق بين php8 والنسخ التي قبلها وما الإضافات التي تم إضافتها على php8 ؟

PHP JIT (مترجم في الوقت المناسب)

الميزة الأكثر شهرة التي تأتي مع PHP 8 هي مترجم JIT (Just-in-time)، ولكن ما هو كل شيء عن JIT؟

RFC JIT يمكن وصفه على النحو التالي:

يتم تنفيذ PHP JIT كجزء شبه مستقل من OPcache، وقد يتم تشغيله / تعطيله في وقت ترجمة PHP وفي وقت التشغيل، فعند التشغيل، يتم تخزين الكود الأصلي لملفات PHP في منطقة إضافية من ذاكرة OPcache المشتركة و opcodes → op_array []. يحتفظ المعالج (المعالجات) بالمؤشرات إلى نقاط إدخال كود JIT-ed. "

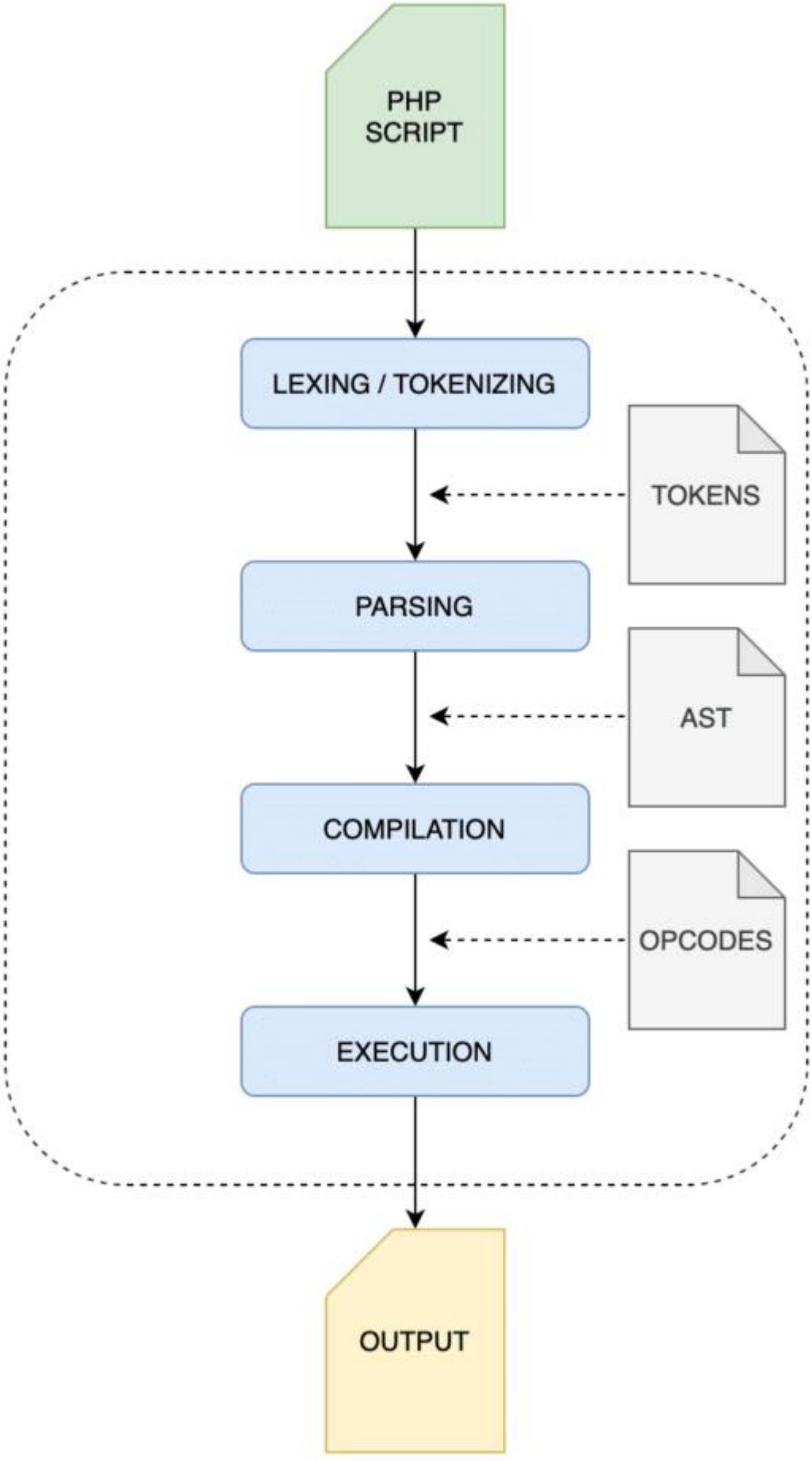
إذن، كيف وصلنا إلى JIT وما الفرق بين JIT و OPcache؟

لفهم ماهية JIT لـ PHP بشكل أفضل، دعنا نلقي نظرة سريعة على كيفية تنفيذ PHP من الكود المصدري إلى النتيجة النهائية.

تنفيذ PHP عبارة عن عملية من ٤ مراحل:

- **التحليل البرمجي أو الترميز Tokenizing:** أولاً يقرأ المترجم كود PHP ويبني مجموعة من الرموز المميزة.
- **التحليل Parsing:** يتحقق المترجم مما إذا كان البرنامج النصي يطابق قواعد بناء الجملة ويستخدم الرموز المميزة لبناء شجرة بناء جملة مجردة (AST)، وهو تمثيل هرمي لهيكل الكود المصدري.
- **التجميع Compilation:** يجتاز المترجم الشجرة ويترجم عقد AST إلى أكواد تشغيل Zend قليل المستوى، وهي عبارة عن معرفات رقمية تحدد نوع التعليمات التي يقوم بها Zend VM.
- **التفسير Interpretation:** يتم تفسير أكواد التشغيل وتشغيلها على Zend VM.

تُظهر الصورة التالية تمثيلاً مرئيًا لعملية تنفيذ PHP الأساسية:


PHP
SCRIPT
LEXING / TOKENIZING
TOKENS
PARSING
AST
COMPILATION
OPCODES
EXECUTION
OUTPUT

إذن كيف تعمل OPcache على جعل PHP أسرع؟ وما هي التغييرات في عملية التنفيذ مع JIT؟

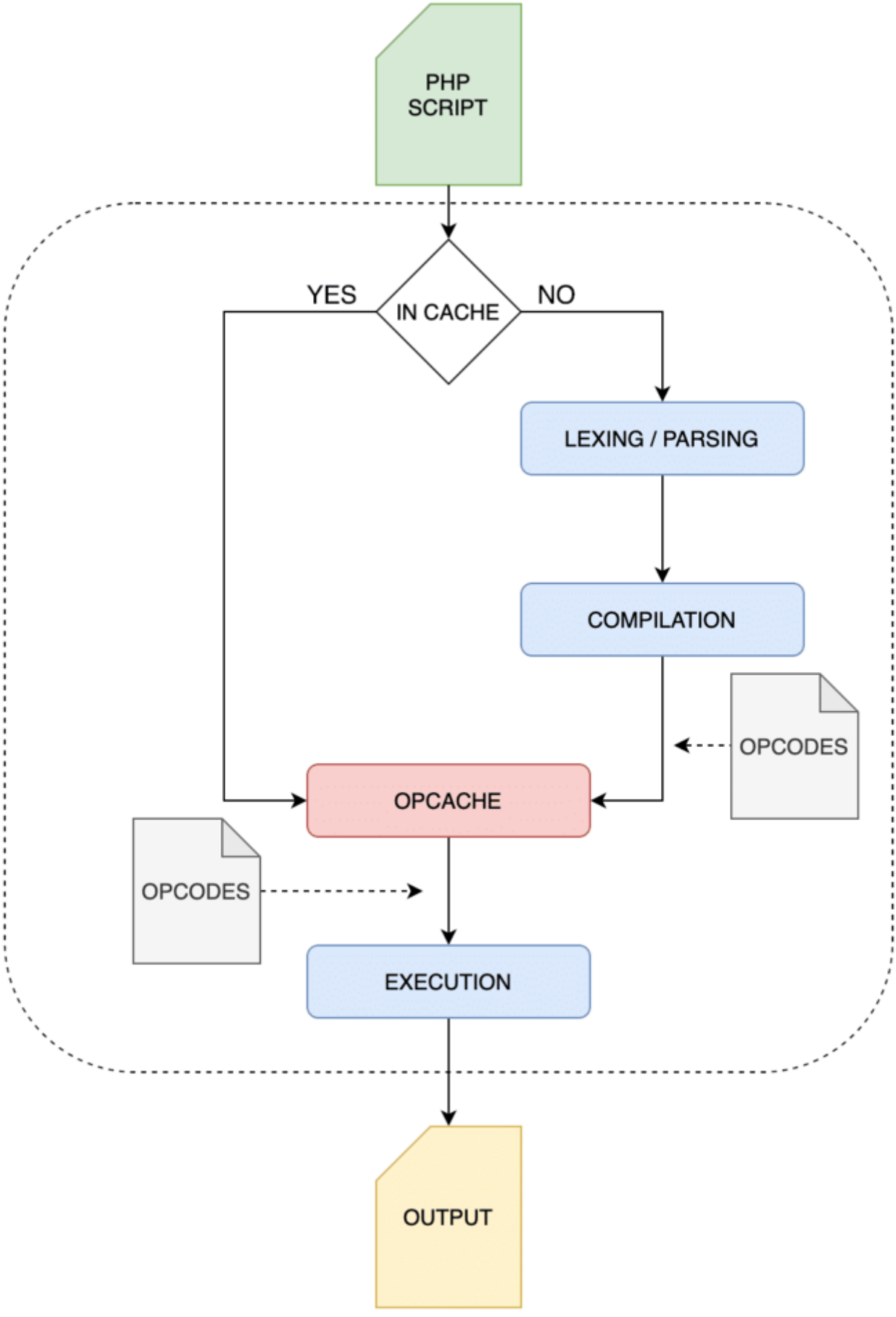
ملحق OPcache

PHP هي لغة مفسرة، وهذا يعني أنه عند تشغيل نص PHP، يقوم المترجم بتحليل الكود وتجميعه وتنفيذه مرارًا وتكرارًا في كل طلب، وقد يؤدي هذا إلى إهدار موارد وحدة المعالجة المركزية والوقت الإضافي.

هذا هو المكان الذي يأتي فيه ملحق OPcache للتشغيل

"يحسِّن OPcache أداء PHP من خلال تخزين كود نص برمجي مترجم مسبقًا في الذاكرة المشتركة؛ وبالتالي عدم الحاجة إلى PHP لتحميل النصوص البرمجية وتحليلها عند كل طلب."

مع تمكين OPcache، يمر مترجم PHP بعملية ٤ مراحل المذكورة بالأعلى فقط في المرة الأولى التي يتم فيها تشغيل البرنامج النصي، وذلك نظرًا لأنه يتم تخزين أكواد PHP bytecodes في الذاكرة المشتركة، فهي متاحة على الفور كتمثيل متوسط قليل المستوى ويمكن تنفيذها على Zend VM على الفور.


PHP
SCRIPT
YES
IN CACHE
NO
LEXING / PARSING
COMPILATION
OPCODES
OPCACHE
OPCODES
EXECUTION
OUTPUT

اعتبارًا من PHP 5.5، يتوفر امتداد Zend OPcache افتراضيًا ويمكنك التحقق مما إذا كان قد تم تكوينه بشكل صحيح عن طريق الاتصالphpinfo() ببساطة من برنامج نصي على الخادم أو التحقق من ملف php.ini.

التحميل المسبق Preloading

تم تحسين OPcache مؤخرًا من خلال تنفيذ التحميل المسبق، وهي ميزة OPcache جديدة تمت إضافتها مع PHP 7.4، حيث يوفر التحميل المسبق طريقة لتخزين مجموعة محددة من البرامج النصية في ذاكرة OPcache " قبل تشغيل أي رمز تطبيق "، ولكنه لا يحقق تحسينًا ملحوظ في الأداء للتطبيقات الويب النموذجية او Web-based applications.

باستخدام JIT ، تتحرك PHP خطوة إلى الأمام.

JIT – مترجم Just in Time

حتى لو كانت أكواد التشغيل في شكل تمثيل متوسط قليل المستوى، فلا يزال يجب تجميعها في كود الآلة، لأن JIT "لا يقدم أي نموذج IR (تمثيل متوسط) إضافي"، ولكنه يستخدم DynASM (مجمع ديناميكي لمحركات توليد الكود) لإنشاء كود أصلي مباشرة من كود بايت PHP.

باختصار، يترجم JIT الأجزاء الساخنة من الكود الوسيط إلى كود الآلة، وبتجاوز التجميع، سيكون قادرًا على إدخال تحسينات كبيرة في الأداء واستخدام الذاكرة.

يوضح زئيف سوراسكي، المؤلف المشارك لاقتراح PHP JIT، مقدار العمليات الحسابية التي ستكون أسرع باستخدام JIT:

ولكن هل ستعمل JIT على تحسين أداء WordPress بشكل فعال؟

JIT لتطبيقات الويب الحية

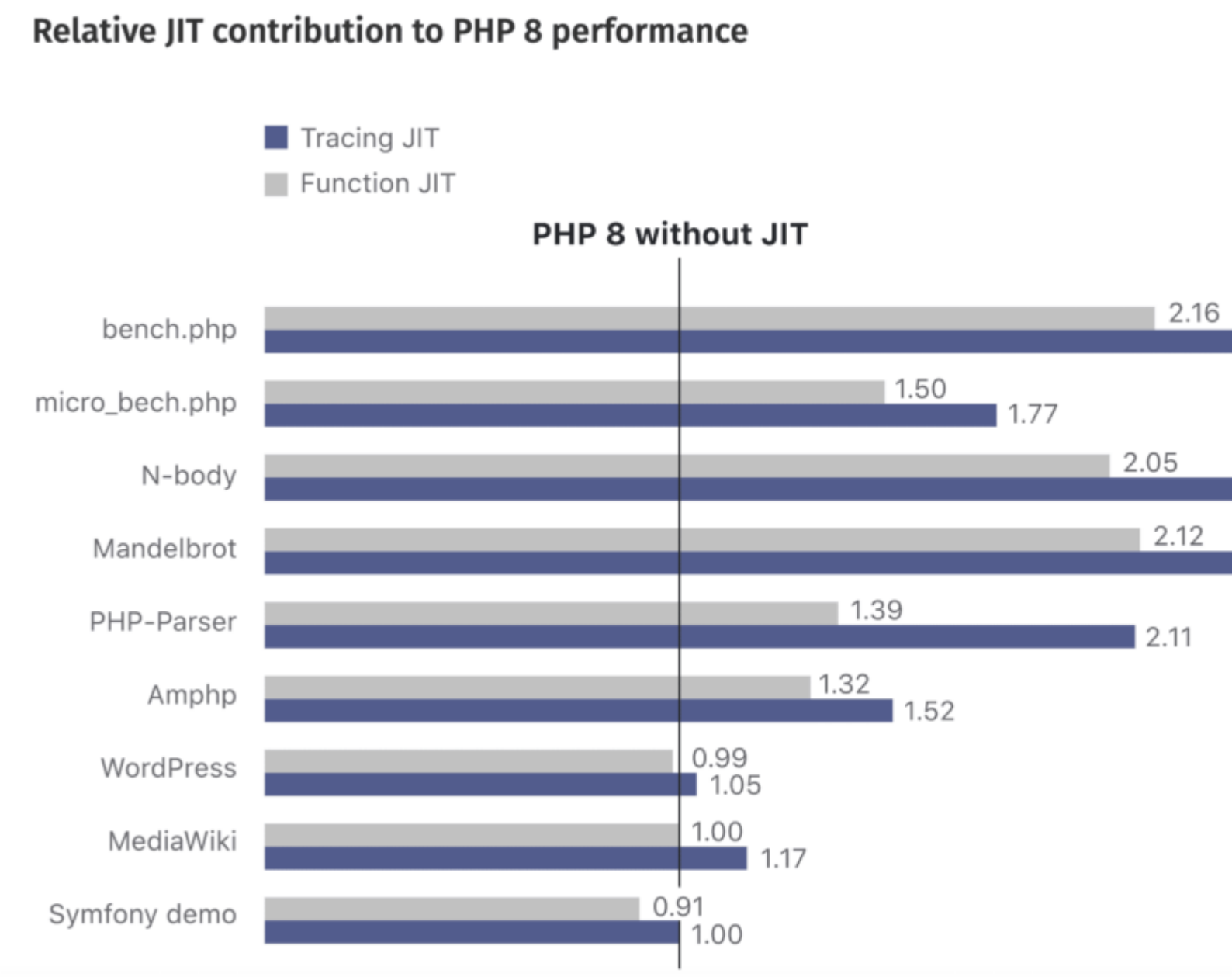
وفقًا لـ JIT RFC، يجب أن يؤدي تطبيق المترجم في الوقت المناسب إلى تحسين أداء PHP، ولكن هل سنشهد حقًا مثل هذه التحسينات في مجلة ادارة المحتوى الشهيرة وردبريس WordPress؟

تظهر الاختبارات المبكرة أن JIT ستجعل أحمال العمل كثيرة الاستخدام لوحدة المعالجة المركزية تعمل بشكل أسرع، ومع ذلك، فإن RFC يحذر من:

"مثل المحاولات السابقة لا يبدو حاليًا أنه يحسن بشكل كبير تطبيقات مثل WordPress (مع opcache.jit = 1235326 req / sec مقابل ٣١٥ req / sec).

من المخطط بذل جهد إضافي، وتحسين JIT لتطبيقات او مجلات ادارة المحتوى، باستخدام تحسينات التنميط والمضاربة ".

مع تفعيل JIT، لن يتم تشغيل الكود بواسطة Zend VM، ولكن بواسطة وحدة المعالجة المركزية نفسها، وهذا من شأنه تحسين السرعة في الحساب، وتطبيقات الويب مثل ووردبريس يعتمد أيضا على عوامل أخرى مثل TTFB، تحسين قاعدة البيانات، طلبات HTTP,, إلى غير ذلك.



PHP 8.0 Announcement Addendum

لذلك عندما يتعلق الأمر بـ WordPress والتطبيقات المماثلة، لا ينبغي أن نتوقع زيادة كبيرة في سرعة تنفيذ PHP، ومع ذلك يمكن أن تحقق JIT العديد من الفوائد للمطورين.

وفقا لنيكيتا بوبوف: مطور برمجيات لدى شركة **JetBrains**

"فوائد مترجم JIT تقريبًا (كما هو موضح بالفعل في RFC):

- أداء أفضل بشكل ملحوظ للرمز الرقمي.
- أداء أفضل قليلاً لكود تطبيق ويب PHP "نموذجي".
- إمكانية نقل المزيد من التعليمات البرمجية من C إلى PHP ، لأن PHP ستكون الآن سريعة بما يكفي ".

لذلك في حين أن JIT بالكاد ستجلب تحسينات كبيرة على أداء WordPress، وستعمل على ترقية PHP إلى المستوى التالي، مما يجعلها لغة يمكن الآن كتابة العديد من الوظائف بها مباشرةً.

ومع ذلك، فإن الجانب السلبي هو التعقيد الأكبر الذي يمكن أن يؤدي إلى زيادة التكاليف في الصيانة والاستقرار وتصحيح الأخطاء، وذلك بحسب ديمتري ستوجوف Core PHP developer :

"JIT بسيط للغاية، ولكنه على أي حال يزيد من مستوى تعقيد PHP بالكامل، ومخاطر ظهور نوع جديد من الأخطاء وتكلفة التطوير والصيانة."

تم تمرير اقتراح تضمين JIT في PHP 8 بأغلبية ٥٠ صوتًا إلى صوتين.

تحسينات PHP 8 وميزات جديدة

بصرف النظر عن JIT، يمكننا توقع العديد من الميزات والتحسينات مع PHP 8:

Constructor Property Promotion

كنتيجة للنقاش المستمر حول كيفية تحسين بيئة العمل للكائن في PHP، يقترح Constructor Property Promotion RFC بنية جديدة وأكثر إيجازًا من شأنها تبسيط إعلان الملكية، مما يجعلها أقل.

حاليًا ، يجب تكرار جميع الخصائص عدة مرات (أربع مرات على الأقل) قبل أن نتمكن من استخدامها مع الكائنات، وخذ بعين الاعتبار المثال التالي من RFC:

```
} Point class

;x$ int public

;y$ int public
```

```
;z$ int public



)construct__ function public

,· = x$ int

,· = y$ int

,· = z$ int

}(

;x$ = x<-this$

;y$ = y<-this$

;z$ = z<-this$

{

{
```

وفقًا لنيكيتا بوبوف، مؤلف RFC، يتعين علينا كتابة اسم الخاصية أربع مرات على الأقل في ثلاثة أماكن مختلفة: إعلان الخاصية، وترويج ملكية المُنشئ، وتخصيص الخاصية، حيث أن بناء الجملة هذا غير قابل للاستخدام بشكل خاص، خاصة في الفصول التي تحتوي على عدد كبير من الخصائص والأسماء الوصفية.

```
} Point class

)construct__ function public

,· = x$ int public

,· = y$ int public

,· = z$ int public

{}(

{
```

يقترح RFC هذا دمج المُنشئ وتعريف الترويج، لذلك اعتبارًا من PHP 8، لدينا طريقة أكثر قابلية للاستخدام للإعلان عن الترويج ويمكن أن يتغير الكود الموضح أعلاه كما هو موضح أدناه:

```
before desugaring //

} Point class

{} (, = x$ int public)construct__ function public

{



after desugaring //

} Point class

;x$ int public



} (, = x$ int)construct__ function public

;x$ = x<-this$

{

{
```

- وهذا كل شيء لذلك لدينا طريقة جديدة للترويج لخصائص أقصر وأكثر قابلية للقراءة وأقل عرضة للأخطاء، وذلك وفقًا لنيكيتا:

إنه تحول نحوي بسيط نقوم به، ولكن هذا يقلل من مقدار الشفرة المعيارية التي يجب عليك كتابتها للأشياء ذات القيمة على وجه الخصوص.

يتم تحويل إعلان الخاصية كما أعلنا صراحةً عن تلك الخصائص ويمكن استخدام واجهة برمجة التطبيقات Reflection لمعرفة تعريفات الخاصية قبل التنفيذ:
التأمل وهو يراقب الحالة بعد التنازل، وهذا يعني أن الخصائص التي تمت ترقيتها ستظهر بنفس طريقة الخصائص المُعلنة صراحةً، وستظهر وسيطات المُنشئ المُروَّجة كوسائط مُنشئ عادية.

الوراثة Inheritance

ليس لدينا أي قيود في استخدام الوراثة بالتزامن مع ملكية المنشئ التي تمت ترقيتها، وعلى أي حال، لا توجد علاقة معينة بين مُنشئي فئة الوالدين والطفل، وذلك وفقا لنيكيتا:

عادة، نقول أن الأساليب يجب أن تكون دائمًا متوافقة مع طريقة الوالدين، لكن هذه القاعدة لا تنطبق على المنشئ، لذا فإن المُنشئ ينتمي حقًا إلى فئة واحدة، ولا يجب أن تكون المُنشئات بين فئة الأصل والطبقة الفرعية متوافقة بأي شكل من الأشكال.

```
} Test class

)construct__ function public

٠ = x$ int public

{} (

{



} Test extends Child class

)construct__ function public

,x$

,٠ = y$ int public

,٠ = z$ int public

} (

;(x$)construct__::parent

{

{
```

ما هو غير مسموح به مع Promoted Properties

يُسمح بالخصائص التي يتم الترويج لها في المنشآت والسمات غير المجردة، ولكن هناك العديد من القيود الجديرة بالذكر هنا.

نبذة مختصرة عن الإنشاء

غير مسموح بالخصائص التي تمت ترقيتها في الفئات والواجهات المجردة:

```
} Test class abstract

.Error: Abstract constructor //

;(x$ private)construct__ function public abstract

{



} Test interface

.Error: Abstract constructor //

;(x$ private)construct__ function public

=class pre><"hcb_wrap"=class div>{

<div/><pre/><code/>{
```

Nullability

واحدة من أبرز القيود المتعلقة بالإلغاء، في السابق، عندما استخدمنا نوعًا لم يكن قابلاً للتقييم بشكل صريح، ولكن بقيمة افتراضية خالية، وكان النوع غير قابل للقيمة ضمنيًا، ولكن مع أنواع الخصائص، ليس لدينا هذا السلوك الضمني لأن المعلمات التي تمت ترقيتها تتطلب إعلانًا عن الخاصية، يجب التصريح عن النوع nullable بشكل صريح.

```
} Test class

Error: Using null default on non-nullable property //

{} (null = prop$ Type public)construct__ function public
```

```
Correct: Make the type explicitly nullable instead //

{} (null = prop$ Type? public)construct__ function public

{
```

النوع القابل للاستدعاء Callable Type

نظرًا لأن Callable ليس نوعًا مدعومًا للخصائص، فلا يُسمح لنا باستخدام النوع القابل للاستدعاء في الخصائص التي تمت ترقيتها:

```
} Test class

.Error: Callable type not supported for properties //

{} (callback$ callable public)construct__ function public

{
```

كلمة var غير مسموح بها

يمكن استخدام كلمة رئيسية فقط مع المعلمات المروجة، لذلك varلا يُسمح بإعلان خصائص المُنشئ بالكلمة الرئيسية (انظر المثال التالي من RFC):

```
} Test class

.Error: "var" keyword is not supported //

{} (prop$ var)construct__ function public

{
```

لا يسمح بالتكرار

يمكننا الجمع بين الخصائص التي تمت ترقيتها والخصائص الصريحة في نفس الفئة، ولكن لا يمكن التصريح عن الخصائص مرتين:

```
} Test class

;prop$ string public

;explicitProp$ int public
```

```
Correct //

} (arg$ int ,promotedProp$ int public)construct__ function public

;arg$ = explicitProp<-this$

{



.Error: Redeclaration of property //

{} (prop$ string public)construct__ function public

{
```

لا يُسمح بالمعلمات المتغيرة

السبب هنا هو أن النوع المُعلن يختلف عن المُعامل المتغير، وهو في الواقع مصفوفة:

```
} Test class

.Error: Variadic parameter //

{} (strings$... string public)construct__ function public

{
```

قراءات إضافية

للحصول على نظرة فاحصة في Costructor Property Promotion، استمع إلى هذه المقابلة مع نيكيتا بوبوف.

التحقق من صحة طرق السمات المجردة

يتم تعريف السمات على أنها "آلية لإعادة استخدام الكود في لغات الوراثة الفردية مثل PHP"، وعادة يتم استخدامها للإعلان عن الطرق التي يمكن استخدامها في فئات متعددة.

يمكن أن تحتوي السمة أيضًا على طرق مجردة، وتعلن هذه الطرق ببساطة عن توقيع الطريقة، لكن تنفيذ الطريقة يجب أن يتم داخل الفصل باستخدام السمة.

وفقًا لدليل PHP،

"تدعم السمات استخدام الأساليب المجردة من أجل فرض متطلبات على فئة العارضين."

هذا يعني أيضًا أن توقيعات الطرق يجب أن تتطابق، وبمعنى آخر، يجب أن يكون نوع وعدد الوسائط المطلوبة هو نفسه.

على أي حال، ووفقًا لنيكيتا بوبوف، مؤلف RFC، لا يتم فرض التحقق من صحة التوقيع حاليًا إلا بشكل مفاجئ:

- لا يتم فرضه في الحالة الأكثر شيوعًا، حيث يتم توفير تنفيذ الطريقة من خلال فئة الاستخدام : https://3v4l.org/SeVK3
- يتم فرضه إذا كان التنفيذ يأتي من فئة رئيسية: https://3v4l.org/4VCIp
- يتم فرضه إذا كان التنفيذ يأتي من فئة فرعية : https://3v4l.org/q7Bq2

المثال التالي من نيكيتا يتعلق بالحالة الأولى (التوقيع غير القسري):

```
} T trait

;(x$ int)test function public abstract

{



} C class

;T use



.Allowed, but shouldn't be due to invalid type //

{} (x$ string)test function public

{
```

مع ذلك يقترح RFC هذا دائمًا عبر خطأ فادح إذا كانت طريقة التنفيذ غير متوافقة مع طريقة السمات المجردة، بغض النظر عن أصلها:

```
with must be compatible (x$ string)test::C of Declaration :error Fatal
١٠ line on test.php/your/to/path/ in (x$ int)test::T
```

تمت الموافقة على RFC هذا بالإجماع.

المصفوفات التي تبدأ بمؤشر سلبي

في PHP، إذا بدأت المصفوفة بمؤشر سالب (start_index < ٠)، فستبدأ المؤشرات التالية من ٠ (المزيد حول هذا في array_fill التوثيق)، ولننظر إلى المثال التالي:

```
;(true ,٤ ,٥-)array_fill = a$

;(a$)var_dump
```

في PHP 7.4، ستكون النتيجة كما يلي:

```
} (٤)array

<=[٥-]

(true)bool

<=[٠]

(true)bool

<=[١]

(true)bool

<=[٢]

(true)bool

{
```

الآن، يقترح RFC هذا تغيير الأشياء بحيث يكون المؤشر الثاني start_index + 1، أيهما قيمة start_index.

في PHP 8 ، ينتج عن الكود أعلاه المصفوفة التالية:

```
} (٤)array

<=[٥-]

(true)bool

<=[٤-]
```

```
(true)bool

<=[٣-]

(true)bool

<=[٢-]

(true)bool

{
```

باستخدام PHP 8، تغير المصفوفات التي تبدأ بمؤشر سلبي سلوكها.

أنواع الاتحاد Union Types 2.0

تقبل أنواع الاتحاد القيم التي يمكن أن تكون من أنواع مختلفة، ولكن حاليًا، لا تقدم PHP دعمًا لأنواع الاتحاد، باستثناء ?Typeبناء الجملة iterable والنوع الخاص .

قبل PHP 8 كان من الممكن تحديد أنواع الاتحاد فقط في تعليقات phpdoc، كما هو موضح في المثال التالي من RFC:

```
} Number class

**/

var int|float $number@ *

/*

;number$ private



**/

param int|float $number@ *

/*

} (number$)setNumber function public

;number$ = number<-this$
```

```
{



**/

return int|float@ *

/*

} ()getNumber function public

;number<-this$ return

{

{
```

الآن ، تقترح أنواع الاتحاد RFC ٢,٠ إضافة دعم لأنواع الاتحاد في توقيعات الوظائف، حتى لا نعتمد على الوثائق المضمنة بعد الآن، ولكننا نحدد أنواع الاتحاد مع |T1|T2، بناء جملة بدلاً من ذلك:

```
} Number class

;number$ float|int private



} void :(number$ float|int)setNumber function public

;number$ = number<-this$

{



} float|int :()getNumber function public

;number<-this$ return

{

{
```

كما أوضح نيكيتا بوبوف في RFC،

“يتيح لنا دعم أنواع الاتحاد في اللغة نقل المزيد من معلومات الكتابة من phpdoc إلى توقيعات الوظائف، مع المزايا المعتادة التي يجلبها هذا:

- يتم فرض الأنواع في الواقع، لذلك يمكن اكتشاف الأخطاء مبكرًا.
- نظرًا لأنه يتم فرضها، فمن غير المرجح أن تصبح معلومات الكتابة قديمة أو تفقد حالاتها.
- يتم فحص الأنواع أثناء الوراثة، وفرض مبدأ استبدال Liskov.
- أنواع متاحة من خلال التفكير.
- بناء الجملة أقل بكثير من لغة pilerplate من phpdoc “.

تدعم أنواع الاتحاد جميع الأنواع المتاحة، مع بعض القيود

- void نوع لا يمكن أن تكون جزءا من الاتحاد، كما void يعني أن وظيفة لا يرجع أي قيمة .
- و null يتم اعتماد نوع فقط في أنواع اتحاد ولكن لا يسمح انها الاستخدام كنوع مستقل.
- ?Tيُسمح أيضًا بتدوين النوع nullable، بمعنى T|null، لكن لا يُسمح لنا بتضمين ?Tالترميز في أنواع الاتحاد (?T1|T2غير مسموح به ويجب أن نستخدمه T1|T2|nullبدلاً من ذلك).
- كما العديد من المهام (أي strpos()، strstr()، substr()، الخ) تشمل falseمن بين أنواع عائد ممكن، و false كما يدعم نوع الزائفة.

أخطاء متسقة من نوع الوظائف الداخلية

عند تمرير معلمة من نوع غير قانوني، والداخلية و المعرفة من قبل المستخدم وظائف تتصرف بشكل مختلف.

تطرح الوظائف المعرفة من قبل المستخدم TypeError، لكن الوظائف الداخلية تتصرف بطرق متنوعة، اعتمادًا على عدة شروط، وعلى أي حال ، فإن السلوك المعتاد هو إلقاء تحذير والعودة null.

```
;((stdClass new)strlen)var_dump
```

قد ينتج عن ذلك التحذير التالي:

```
in object given ,string be to \ expects parameter ()strlen :Warning
ٔ line on test.php/your/to/path/

NULL
```

إذا strict types تم تفعيله، أو تحدد معلومات الوسيطة أنواعًا، فسيكون السلوك مختلفًا، في مثل هذه السيناريوهات، يتم اكتشاف خطأ النوع وينتج عنه ملف TypeError.

قد يؤدي هذا الموقف إلى عدد من المشكلات الموضحة جيدًا في قسم مشكلات RFC .

لإزالة هذه التناقضات، يقترح RFC هذا جعل المعلمة الداخلية لتحليل واجهات برمجة التطبيقات لإنشاء دائمًا Throw Error في حالة عدم تطابق نوع المعلمة.
في PHP 8 ، يظهر الرمز أعلاه الخطأ التالي:

التخلص من التعبير

في PHP، throw هو بيان، لذلك فمن غير الممكن استخدامها في الأماكن التي يكون فيها سوى التعبير مسموح به.

يقترح RFC هذا تحويل throw العبارة إلى تعبير بحيث يمكن استخدامه في أي سياق يُسمح فيه بالتعبيرات، وعلى سبيل المثال، السهم، والوظائف، مشغل تتجمع لاغيا، الثلاثي والفيس المشغلين، الخ.

انظر الأمثلة التالية من RFC:

```
;()Exception new throw <= ()fn = callable$



.value is non-nullable$ //

;()InvalidArgumentException new throw ?? nullableValue$ = value$



.value is truthy$ //

;()InvalidArgumentException new throw :? falsableValue$ = value$
```

خرائط ضعيفة

الخريطة الضعيفة هي مجموعة بيانات (كائنات) يتم الإشارة فيها إلى المفاتيح بشكل ضعيف، مما يعني أنه لا يتم منعها من جمع البيانات غير الهامة.

أضاف PHP 7.4 دعمًا للمراجع الضعيفة كطريقة للاحتفاظ بمرجع إلى كائن لا يمنع تدمير الكائن نفسه. كما أشار نيكيتا بوبوف،

"المراجع الضعيفة الخام ليست سوى ذات فائدة محدودة في حد ذاتها، والخرائط الضعيفة أكثر شيوعًا في الممارسة العملية، حيث لا يمكن تنفيذ خريطة ضعيفة فعالة فوق مراجع PHP الضعيفة لأن القدرة على تسجيل رد اتصال التدمير غير متوفرة ".

لهذا السبب تقدم RFCWeakMap فئة لإنشاء كائنات لاستخدامها كمفاتيح خرائط ضعيفة يمكن تدميرها وإزالتها من الخريطة الضعيفة إذا لم تكن هناك أي إشارات أخرى إلى الكائن الرئيسي.

في العمليات طويلة المدى، سيمنع هذا تسرب الذاكرة وتحسين الأداء. انظر المثال التالي من RFC:

```
;WeakMap new = map$

;stdClass new = obj$

;٤٢ = [obj$]map$

;(map$)var_dump
```

باستخدام PHP 8 ، ينتج عن الكود أعلاه النتيجة التالية (انظر التعليمات البرمجية قيد التشغيل هنا):

```
} (١) ١#(WeakMap)object

<=[٠]

} (٢)array

<=["key"]

} (٠) ٢#(stdClass)object

{

<=["value"]

(٤٢)int

{

{
```

إذا قمت بإلغاء تحديد الكائن، فسيتم إزالة المفتاح تلقائيًا من الخريطة الضعيفة:

```
;(obj$)unset

;(map$)var_dump
```

الآن ستكون النتيجة كما يلي:

```
} (٠) ١#(WeakMap)object
```

```
{
```

لإلقاء نظرة فاحصة على الخرائط الضعيفة، راجع RFC، فلقد تمت الموافقة على الاقتراح بالإجماع.

فاصلة زائدة في قائمة المعلمات

يتم إلحاق الفاصلات اللاحقة بقوائم العناصر في سياقات مختلفة، فقد قدمت PHP 7.2 فواصل لاحقة في بناء جملة القائمة، بينما قدمت PHP 7.3 فواصل لاحقة في استدعاءات الوظائف.

تقدم PHP 8 الآن فاصلات لاحقة في قوائم المعلمات بالوظائف والطرق والإغلاق، كما هو موضح في المثال التالي:

```
} Foo class

)construct__ function public

,x$ string

,y$ int

trailing comma // ,z$ float

}(

do something //

{

{
```

تم تمرير هذا الاقتراح بأغلبية ٥٨ صوتًا مقابل ١.

السماح ببناء جمل الكلاسات داخل الكائنات

من أجل جلب اسم الفصل، يمكننا استخدام Foo\Bar::class بناء الجملة، يقترح RFC بتمديد نفس بناء الجملة للكائنات بحيث يمكن الآن جلب اسم فئة كائن معين كما هو موضح في المثال أدناه:

```
;stdClass new = object$

"stdClass" // ;(class::object$)var_dump
```

```
;null = object$

TypeError // ;(class::object$)var_dump
```

تمت الموافقة على هذا الاقتراح بالإجماع.

سمات v2 Attributes

السمات، المعروفة أيضًا باسم التعليقات التوضيحية، هي شكل من أشكال البيانات الوصفية المهيكلة التي يمكن استخدامها لتحديد خصائص الكائنات أو العناصر أو الملفات.

حتى PHP 7.4، كانت تعليقات المستندات هي الطريقة الوحيدة لإضافة البيانات الوصفية إلى إعلانات الفئات والوظائف وما إلى ذلك، ولكن الآن، تقدم Attributes v2 RFC سمات لـ PHP لتعريفها كشكل من البيانات الوصفية الهيكلية والنحوية التي يمكن إضافتها إلى إعلانات الفئات والخصائص والوظائف والطرق والمعلمات والثوابت.

تُضاف السمات قبل الإعلانات التي تشير إليها، فقط انظر الأمثلة التالية من RFC:

```
<<ExampleAttribute>>

Foo class

}

<<ExampleAttribute>>

;'foo' = FOO const public



<<ExampleAttribute>>

;x$ public



<<ExampleAttribute>>

{ } (bar$ <<ExampleAttribute>>)foo function public

{
```

```
;{ } () class <<ExampleAttribute>> new = object$



<<ExampleAttribute>>

{ } ()f1 function



;{ } () function <<ExampleAttribute>> = f2$



;١ <= () fn <<ExampleAttribute>> = f3$
```

يمكن إضافة السمات قبل تعليق حظر المستند أو بعده:

```
<<ExampleAttribute>>

/* docblock **/

<<AnotherExampleAttribute>>

{} ()foo function
```

يمكن أن يحتوي كل إعلان على سمة واحدة أو أكثر وقد تحتوي كل سمة على قيمة مرتبطة واحدة أو أكثر:

```
<<WithoutArgument>>

<<(٠)SingleArgument>>

<<('World','Hello')FewArguments>>

{} ()foo function
```

Arguments المسماة

توفر الوسيطات المسماة طريقة جديدة لتمرير الوسائط إلى دالة في PHP:

تسمح الوسيطات المسماة بتمرير الوسائط إلى وظيفة بناءً على اسم المعلمة، بدلاً من موضع المعلمة.

يمكننا تمرير الوسائط المسماة إلى دالة عن طريق إضافة اسم المعلمة قبل قيمتها:

```
callFunction(name: $value);
```

يُسمح لنا أيضًا باستخدام الكلمات الرئيسية المحجوزة، كما هو موضح في المثال أدناه:

```
callFunction(array: $value);
```

لكن لا يُسمح لنا بتمرير اسم المعلمة ديناميكيًا، بل يجب أن تكون المعلمة معرّفًا ولا يُسمح بالصياغة التالية:

```
callFunction($name: $value);
```

وفقًا لنيكيتا بوبوف، مؤلف RFC، تقدم الحجج المسماة العديد من المزايا.

أولاً ستساعدنا الوسيطات المسماة على كتابة كود أكثر قابلية للفهم لأن معناها هو التوثيق الذاتي.
المثال أدناه من RFC واضح بذاته:

```
array_fill(start_index: ٠, num: ١٠٠, value: ٥٠);
```

الحجة المسماة مستقلة عن النظام، وهذا يعني أننا لسنا مجبرين على تمرير الحجج إلى وظيفة بنفس ترتيب توقيع الوظيفة:

```
array_fill(value: ٥٠, num: ١٠٠, start_index: ٠);
```

من الممكن أيضًا دمج الوسائط المسماة مع الوسائط الموضعية:

```
htmlspecialchars($string, double_encode: false);
```

ميزة أخرى رائعة للوسيطات المسماة أنها تسمح فقط بتحديد تلك الوسيطات التي نريد تغييرها بالفعل ولا يتعين علينا تحديد الوسائط الافتراضية إذا كنا لا نريد الكتابة فوق القيم الافتراضية، ويوضح المثال التالي من RFC:

```
htmlspecialchars($string, default, default, false);

// vs

htmlspecialchars($string, double_encode: false);
```

هام:

إذا كنت مطور WordPress، فيرجى ملاحظة أنه في وقت كتابة هذا التقرير، قد تؤدي الوسائط المسماة إلى مشكلات التوافق مع الإصدارات السابقة، لذلك لا تستخدمها في الإنتاج بدون اختبار مدروس.

يمكن استخدام الوسائط المسماة مع سمات PHP ، كما هو موضح في المثال التالي من RFC:

```
<<('B' :b ,'A')MyAttribute>>

{} Test class
```

ومع ذلك، لا يُسمح بتمرير الوسائط الموضعية بعد الوسائط المسماة وسيؤدي ذلك إلى حدوث خطأ في وقت الترجمة، ويحدث الشيء نفسه عند تمرير نفس اسم المعلمة مرتين.

تعتبر الوسيطات المسماة مفيدة بشكل خاص مع إعلانات الفئة لأن المنشئات عادةً ما يكون لديها عدد كبير من المعلمات وتوفر الوسائط المسماة طريقة أكثر "مريحة" للإعلان عن فئة.

Nullsafe

يقدم هذا RFC المشغل nullsafe بتقييم $->كامل للدارة القصيرة.

في تقييم الدارة القصيرة، يتم تقييم العامل الثاني فقط إذا لم يقم المشغل الأول بتقييمه null، وإذا قام عامل في سلسلة بالتقييم لـ null، فإن تنفيذ السلسلة بأكملها يتوقف ويقيم null.

خذ بعين الاعتبار الأمثلة التالية من RFC:

```
;()b<-?a$ = foo$
```

إذا كانت القيمة a$ خالية ، b()فلن يتم استدعاء الطريقة foo$ ويتم تعيينها على null.

سلسلة Saner لمقارنات الأرقام

في إصدارات PHP السابقة، عند إجراء مقارنة غير صارمة بين السلاسل والأرقام، تقوم PHP أولاً بنقل السلسلة إلى رقم، ثم تقوم بإجراء المقارنة بين الأعداد الصحيحة أو العائمة، حتى إذا كان هذا السلوك مفيدًا جدًا في العديد من السيناريوهات، فقد ينتج عنه نتائج خاطئة قد تؤدي أيضًا إلى أخطاء أو مشكلات أمنية.

خذ بعين الاعتبار المثال التالي من RFC:

```
;["baz","bar","foo"] = validValues$

;'' = value$
```

```
;((validValues$ ,value$)in_array)var_dump

bool(true) //
```

تقدم PHP 8 سلسلة Saner إلى مقارنات بين الأرقام ، بهدف جعل المقارنات بين الأعداد أكثر منطقية، على حد تعبير نيكيتا بوبوف:

يهدف RFC هذا إلى إعطاء مقارنات سلسلة إلى رقم سلوكًا أكثر منطقية: عند المقارنة بسلسلة رقمية، استخدم مقارنة رقم (كما هو الحال الآن)، وعلى خلاف ذلك، قم بتحويل الرقم إلى سلسلة واستخدم مقارنة السلسلة.

يقارن الجدول التالي سلوك السلسلة بأرقام مقارنة إصدارات PHP السابقة وفي PHP 8:

```
After|Before|  Comparison

---------------------------

true| true|  "٠" == ٠

true| true| "٠٫٠" == ٠

false| true| "foo" == ٠

false| true|  "" == ٠

true| true|"٤٢ " == ٤٢

false| true|"foo٤٢" == ٤٢
```

سلاسل سانر رقمية Saner Numeric Strings

في PHP ، تنقسم السلاسل التي تحتوي على أرقام إلى ثلاث فئات :

- سلاسل رقمية : سلاسل تحتوي على رقم تسبقه مسافات بشكل اختياري.
- السلسلة الرقمية البادئة : السلاسل التي تكون أحرفها الأولية عبارة عن سلاسل رقمية والأحرف اللاحقة تكون غير رقمية.
- سلسلة غير رقمية : سلاسل لا تقع في أي من الفئات السابقة.

يتم التعامل مع السلاسل الرقمية والسلاسل الرقمية البادئة بشكل مختلف بناءً على العملية التي يتم إجراؤها. فمثلا:

- سلسلة واضحة لتحويل عدد (أي (int)و (float)نوع يلقي) الرقمية وتحويل أرقام سلاسل الرائدة رقمية، وتحويل سلسلة غير رقمية بشكل صريح إلى رقم ينتج ٠ .

- تؤدي السلسلة الضمنية إلى تحويلات الأرقام (أي عدم strict type الإعلان) إلى نتائج مختلفة للسلاسل الرقمية وغير الرقمية. سلسلة غير رقمية للتحويلات الرقمية تطرح ملف TypeError.
- is_numeric()إرجاع صحيح فقط للسلاسل الرقمية.

تؤدي إزاحة السلاسل، والعمليات الحسابية، وعمليات الزيادة والنقصان، والمقارنات من سلسلة إلى سلسلة، والعمليات الأحادية أيضًا إلى نتائج مختلفة.

يقترح طلب التعليقات هذا

توحيد أوضاع السلسلة الرقمية المختلفة في مفهوم واحد: الأحرف الرقمية فقط مع السماح بالمسافة البيضاء البادئة واللاحقة، وأي نوع آخر من السلاسل هو غير رقمي وسيرمي TypeErrors عند استخدامه في سياق رقمي.

هذا يعني أن جميع السلاسل التي تصدر حاليًا E_NOTICE "تمت مصادفة قيمة رقمية غير جيدة التكوين" ستتم إعادة تصنيفها في E_WARNING "تمت مصادفة قيمة غير رقمية" إلا إذا احتوت السلسلة الرقمية البادئة على مسافة بيضاء لاحقة فقط، وستتم ترقية الحالات المختلفة التي تصدر حاليًا رسالة تحذير إلى TypeErrors.

للحصول على نظرة عامة أعمق على السلاسل الرقمية في PHP 8، مع أمثلة التعليمات البرمجية والاستثناءات ومشكلات التوافق مع الإصدارات السابقة ، راجع RFC .

مطابقة التعبير v2.0

يتشابه التعبير الجديد match إلى حد كبير switch ولكن مع الدلالات الأكثر أمانًا ولكن مع السماح بإرجاع القيم.

لفهم الفرق بين بنيتي التحكم، ضع في اعتبارك switch المثال التالي من RFC :

```
switch (1) {
    case 0:
        $result = 'Foo';
        break;
    case 1:
        $result = 'Bar';
        break;
```

```
:٢ case

;'Baz' = result$

;break

{



;result$ echo

Bar <//
```

يمكننا الآن الحصول على نفس نتيجة الكود أعلاه match بالتعبير التالي :

```
} (١) match echo

,'Foo' <= ٠

,'Bar' <= ١

,'Baz' <= ٢

;{

Bar <//
```

الميزة الكبيرة لاستخدام التعبير match الجديد هي أنه بينما switch يقارن القيم بشكل فضفاض (==) يحتمل أن يؤدي إلى نتائج غير متوقعة، مع matchالمقارنة هي التحقق من الهوية (===).

و تعبير match يحتوي أيضا عدة عبارات مفصولة بفواصل السماح لمزيد من جملة موجزة (مصدر):

```
} (x$) match = result$

:This match arm //

,٥ <= c$ ,b$ ,a$

:Is equivalent to these three match arms //

,٥ <= a$
```

```
,٥ <= b$

,٥ <= c$

;{
```

عمليات التحقق من نوع أكثر صرامة لعمليات حسابية / على مستوى البت

في الإصدارات السابقة من PHP، وتطبيق الحسابية و المختصة بالبت المشغلين لمجموعة، الموارد، على أي حال، كان السلوك في بعض الأحيان غير متسق.

في هذا RFC، يوضح نيكيتا بوبوف كيف يمكن أن يكون هذا السلوك غير معقول بمثال بسيط:

```
;([٤٢] % [])var_dump

(٠)int //
```

يشرح نيكيتا كيف أدى تطبيق عامل حسابي أو أحادي على المصفوفات أو الموارد أو الكائنات غير المحملة بشكل زائد إلى نتائج مختلفة :

عوامل التشغيل +، -، *، /، **:

- طرح استثناء خطأ في معامل الصفيف. (باستثناء + إذا كان كلا المعاملين مصفوفة.)
- بصمت تحويل معامل المورد إلى معرّف المورد كعدد صحيح.
- تحويل معامل الكائن إلى عدد صحيح واحد ، أثناء إلقاء إشعار.

عوامل التشغيل٪ ، << ، >> ، & ، | ، ^:

- قم بتحويل معامل الصفيف بصمت إلى عدد صحيح صفر إذا كان فارغًا أو إذا كان عددًا صحيحًا واحدًا إذا لم يكن فارغًا.
- بصمت تحويل معامل المورد إلى معرّف المورد كعدد صحيح.
- تحويل معامل الكائن إلى عدد صحيح واحد ، أثناء إلقاء إشعار.

عامل التشغيل ~:

- قم بطرح استثناء خطأ لمعاملات المصفوفة والموارد والكائن.

عوامل التشغيل ++ و -:

- لا تفعل شيئًا بصمت إذا كان المعامل عبارة عن مصفوفة أو مورد أو كائن.

مع PHP 8، تتغير الأشياء ويظل السلوك هو نفسه لجميع العمليات الحسابية ومعاملات البت:

قم بطرح TypeError استثناء لمعاملات المصفوفة والموارد والكائنات.

وظائف PHP جديدة

يأتي PHP 8 بالعديد من الوظائف الجديدة إلى اللغة:

str_contains

قبل PHP 8، كانت strstr و strpos هي الخيارات النموذجية للمطورين للبحث عن إبرة داخل سلسلة معينة، والمشكلة هي أن كلتا الوظيفتين لا تعتبر بديهية للغاية ويمكن أن يكون استخدامهما مربكًا لمطوري PHP الجدد. انظر المثال التالي:

```
;'Managed WordPress Hosting'=mystring$

;'WordPress'=findme$

;(findme$,mystring$)strpos=pos$



}(false==!pos$)if

;"The string has been found"echo

}else{

;"String not found"echo

{
```

في المثال أعلاه، استخدمنا !==عامل المقارنة، والذي يتحقق أيضًا مما إذا كانت القيمتان من نفس النوع، وهذا يمنعنا من الحصول على خطأ إذا كان موضع الإبرة ٠ :

"قد ترجع هذه الدالة Boolean FALSE، ولكنها قد ترجع أيضًا قيمة غير منطقية يتم تقييمها إلى FALSE . [...] استخدم عامل التشغيل === لاختبار قيمة الإرجاع لهذه الوظيفة. "

علاوة على ذلك، توفر العديد من الأطر وظائف مساعدة للبحث عن قيمة داخل سلسلة معينة.

الآن، هذا RFC يقترح استحداث وظيفة جديدة تسمح للبحث داخل سلسلة: str_contains.

```
bool:(needle$string,haystack$string)str_contains
```

استخدامه بسيط جدا. str_contains يتحقق مما إذا needle$ تم العثور عليه haystack$ وإرجاعه true أو falseوفقًا لذلك.

لذلك، بفضل str_contains، يمكننا كتابة الكود التالي:

```
;'DEEPTECH WordPress' = mystring$

;'WordPress' =  findme$



} ((findme$ ,mystring$)str_contains) if

;"The string has been found" echo

} else {

;"String not found" echo

{
```

أيهما أكثر قابلية للقراءة وأقل عرضة للأخطاء.

في وقت كتابة هذه السطور، كانت str contains حساسة لحالة الأحرف، ولكن هذا قد يتغير في المستقبل.

str_starts_with () و str_ends_with ()

بالإضافة إلى وظيفة str_contains، تسمح وظيفتان جديدتان بالبحث عن إبرة داخل سلسلة معينة: str_starts_with و str_ends_with.

تتحقق هذه الوظائف الجديدة مما إذا كانت سلسلة معينة تبدأ أو تنتهي بسلسلة أخرى:

```
bool : (needle$ string , haystack$ string) str_starts_with

bool : (needle$ string , haystack$ string) str_ends_with
```

تعود كلتا الوظيفتين false إذا كان needle$ أطول من haystack$.

وفقًا لـ Will Hudgins، مؤلف RFC هذا:

"إن str_starts_with(وstr_ends_with الوظيفة مطلوب بشكل كبير لدرجة أن العديد من أطر PHP الرئيسية تدعمها، بما في ذلك في symfony، لارافل، – Yii،FuelPHP، و فالكون ".

بفضلهم يمكننا الآن تجنب استخدام وظائف دون المستوى وأقل بديهية مثل substr، strpos. كلتا الوظيفتين حساسة لحالة الأحرف:

```
;"WordPress" = str$

;"!Found" echo (("Word" ,str$)str_starts_with) if



;"!Not  found" echo (("word" ,str$)str_starts_with) if
```

get_debug_type

get_debug_type هي دالة PHP جديدة تُرجع نوع المتغير، وتعمل الوظيفة الجديدة بطريقة مشابهة تمامًا gettypeللوظيفة ، ولكنها get_debug_typeتُرجع أسماء الأنواع الأصلية وتحل أسماء الفئات.

يعد هذا تحسينًا جيدًا للغة، حيث gettype()إنه غير مفيد للتحقق من الكتابة.

يقدم RFC مثالين مفيدين لفهم الفرق بين get_debug_type()الوظيفة الجديدة و gettype()، والمثال الأول يظهر gettype في العمل:

```
;[١,٢,٣] = bar$



} ((Foo instanceof bar$)!) if

? (bar$)is_object) . 'got ,' . class::Foo . 'Expected')TypeError new throw
;(((bar$)gettype : (bar$)get_class

{
```

باستخدام PHP 8، يمكننا استخدام get_debug_type:

```
} ((Foo instanceof bar$)!) if

. 'got' . class::Foo . 'Expected')TypeError new throw
;((bar$)get_debug_type

{
```

يوضح الجدول التالي قيم الإرجاع لـ get_debug_typeو gettype:

Value	gettype()	get_debug_type()
١	integer	int
٠,١	double	float
true	boolean	bool
false	boolean	bool
null	NULL	null
”WordPress“	string	string
[١,٢,٣]	array	array
”A class with name “Foo\Bar	object	Foo\Bar
An anonymous class	object	class@anonymous

طلبات التعليقات الإضافية

فيما يلي قائمة سريعة بالتحسينات الإضافية المعتمدة القادمة مع PHP 8:

- واجهة Stringable: تقدم RFC واجهة Stringable تتم إضافتها تلقائيًا إلى الفئات التي تطبق to String()الطريقة، والهدف الرئيسي هنا هو استخدام string|String Table نوع الاتحاد.
- واجهات برمجة التطبيقات الجديدة لـ DOM Living Standard في ext / dom: يقترح RFC هذا تنفيذ معيار المعيشة DOM الحالي إلى امتداد PHP DOM من خلال تقديم واجهات جديدة وخصائص عامة.
- ثابت نوع الإرجاع: PHP 8 يدخل استخدام static كنوع عودة بجانب self parent أنواع.
- تعديلات بناء الجملة المتغيرة: RFC هذا يحل بعض التناقضات المتبقية في بناء الجملة المتغير لـ PHP.