

Artigo

[Andre Larsen Barbosa](#) · Jul. 8, 2021 · 11 min de leitura

Apresentando novos recursos JSON no Caché 2016.1

Esta postagem tem como objetivo guiá-lo através dos novos recursos JSON que introduzimos no Caché 2016.1. JSON surgiu para um formato de serialização usado em muitos lugares. A web começou, mas hoje é utilizada em todos os lugares. Temos muito o que abordar, então vamos começar.

Aviso: alguns dos recursos e sintaxe documentados aqui foram posteriormente alterados em 2016.2, portanto, só funcionam em 2016.1. Veja este [outro artigo](#) para detalhes.

Nosso suporte JSON até agora estava fortemente acoplado ao framework ZEN, nosso próprio framework para construção de aplicações web. Com o tempo, reconhecemos uma demanda crescente para acessar a funcionalidade JSON fora do ZEN. Embora fosse possível ser usado fora do ZEN, era confuso e o antigo suporte JSON incluía algumas deficiências que exigiam a introdução de uma nova API. Eu quero descrever a nova API nesta postagem e fornecer algumas informações básicas por que fizemos certas escolhas de design.

A primeira mudança que você notará é a introdução de duas novas classes, `%Object` e `%Array`, ambas estendendo `%AbstractObject`. Essas classes são sua interface primária para lidar com dados dinâmicos, ou seja, dados com uma estrutura dinâmica como JSON. Ambas as classes estão localizadas no pacote `%Library` conforme implementam a funcionalidade de uso geral.

`%Object` permite que você crie o que chamamos de objeto dinâmico. Você não implementará uma subclasse, mas sim instanciará um objeto dinâmico e adicionará as propriedades necessárias no tempo de execução. Você também pode remover propriedades e fornecemos uma interface iteradora para introspecção e descoberta de propriedades existentes. As propriedades estão sempre desordenadas e não garantimos nenhum pedido.

```
USER>set object = ##class(%Object).$new()  
USER>set object.name = "Stefan Wittmann"  
USER>set object.lastSeriesSeen = "Daredevil"  
USER>set object.likes = "Galaxy"
```

O exemplo de código acima cria um novo objeto dinâmico com três propriedades `name`, `lastSeriesSeen` e `likes`.

`%Array` é uma coleção ordenada de valores. Você pode adicionar novos valores a uma matriz, pode removê-los e iterar sobre ela. Eu acho que você conseguiu a foto. Chamamos essas matrizes dinâmicas. Os arrays suportam dispersão, o que significa que você pode atribuir um valor ao slot 100, enquanto os slots 0 a 99 são deixados sem atribuição e alocamos espaço apenas para um valor, em vez de 101.

```
USER>set array = ##class(%Array).$new()  
USER>do array.$push(1)  
USER>do array.$push("This is a string")  
USER>do array.$push(object)
```

O exemplo acima cria uma matriz dinâmica com três valores. O primeiro é um número, o segundo é uma string e o último é o objeto dinâmico do nosso exemplo anterior. Estamos construindo uma matriz dinâmica densa, inserindo novos valores na matriz. Uma matriz esparsa pode ser construída definindo valores em uma chave específica. Mas vamos guardar a dispersão para mais tarde.

Produzindo JSON

Se você quiser serializar uma entidade dinâmica para JSON, basta chamar o método `$toJSON()` nela. O método é muito inteligente e retorna uma string se ele se ajustar a uma única string e um objeto de fluxo caso contrário. Se for usado em uma instrução `DO`, ele enviará a saída para o dispositivo atual. A saída será atribuída à variável do lado esquerdo se for usada em uma instrução `SET`.

```
USER>do object.$toJSON()
{"name":"Stefan Wittmann","lastSeriesSeen":"Daredevil","likes":"Galaxy"}
USER>do array.$toJSON()
[1,"This is a string.",{"name":"Stefan Wittmann","lastSeriesSeen":"Daredevil","likes":"Galaxy"}]
```

Consumindo JSON

A outra direção também é bastante simples. Existem várias maneiras de receber um objeto JSON, mas quase sempre ele terminará em uma variável, seja como uma string ou um fluxo, dependendo do tamanho. Basta chamar `$fromJSON()` e passar sua string JSON ou stream. Nós cuidaremos do resto.

```
USER>set someJSONstring = '{"firstname":"Stefan","lastname":"Wittmann"}'
USER>set consumedJSON = ##class(%AbstractObject).$fromJSON(someJSONstring)
USER>write consumedJSON.$size()
2
USER>write consumedJSON.$toJSON()
{"firstname":"Stefan","lastname":"Wittmann"}
```

O exemplo de código acima usa o método `$size()` para obter o número de propriedades do objeto `consumedJSON`. `$size()` também está disponível para matrizes e retorna o número de valores atribuídos na matriz.

Movendo

Agora que você viu os fundamentos da nova API para objetos e matrizes dinâmicos, vamos explorar alguns dos recursos e tópicos mais avançados. Como as estruturas de dados são definidas em tempo de execução, é muito importante fornecer ferramentas adequadas para descobrir o conteúdo. O utilitário mais importante é um iterador que permite fazer um loop pelas propriedades de um objeto dinâmico.

```
USER>set iter = object.$getIterator()
USER>while iter.$getNext(.key,.value) { write "key "_key_":_value,! }
key name:Stefan Wittmann
key lastSeriesSeen:Daredevil
key likes:Galaxy
```

Um aspecto muito importante ao projetar a API foi a consistência. Uma funcionalidade genérica como um iterador deve se comportar da mesma forma para objetos e matrizes.

```
USER>set iter = array.$getIterator()
USER>while iter.$getNext(.key,.value) { write "key "_key_":_value,! }
key 0:1
```

```
key 1:This is a string.  
key 2:2@%Library.Object
```

O iterador nos permite introspectar facilmente o conteúdo de uma matriz e, portanto, podemos agora discutir matrizes esparsas. Vamos supor que definimos outro valor no índice posicional 10. Lembre-se de que as matrizes são baseadas em zero.

```
USER>do array.$set(10,"This is a string in a sparse array")  
USER>write array.$toJSON()  
[1,"This is a string.",{"name":"Stefan Wittmann","lastSeriesSeen":"Daredevil","likes":  
:"Galaxy"},null,null,null,null,null,null,null,"This is a string in a sparse array"]
```

Você pode observar que o 11º valor é definido para a string “ Esta é uma string em uma matriz esparsa ” e todos os slots entre os índices 3 e 9 são serializados com valores nulos. Esses valores nulos não existem na memória e são serializados apenas como valores nulos porque JSON não oferece suporte a valores indefinidos. Você pode provar isso facilmente iterando sobre a matriz como o seguinte trecho de código.

```
USER>set iter = array.$getIterator()  
USER>while iter.$getNext(.key,.value) { write "key "_key_":_value,! }  
key 0:1  
key 1:This is a string.  
key 2:2@%Library.Object  
key 10:This is a string in a sparse array
```

Como você pode ver, apenas 4 chaves estão definidas e as chaves 3 a 9 não estão configuradas para nenhum valor. O exemplo de código demonstra esse conceito com uma matriz, mas o mesmo é verdadeiro para objetos dinâmicos. É muito importante para vários ambientes lidar com dados esparsos com eficiência e você verá algumas referências a isso em postagens de blog que compartilharei mais tarde.

Tratamento de erro e algum doce

Outro fato importante é que estamos lançando exceções no caso de um erro, em vez de retornar um valor de %Status. Vamos ver o que acontece se tentarmos analisar uma string JSON inválida.

```
USER>set invalidObject = ##class(%AbstractObject).$fromJSON("{,}")  
<THROW>zfromJSON+24^%Library.AbstractObject.1 *%Exception.General Parsing error 3 Lin  
e 1 Offset 2
```

Você pode ver que a exceção lançada inclui informações suficientes para concluir que o segundo caractere na primeira linha é inválido. Portanto, qualquer código que faz uso da nova API JSON deve ser cercado por um bloco try / catch em algum nível. Se você pensar bem, isso faz sentido, pois estamos lidando com dados dinâmicos e os dados podem não se adequar às suas suposições.

Existem vários benefícios em usar exceções para o mecanismo de relatório, mas o motivo mais importante é que ele permite que cada método retorne uma referência à saída, permitindo, portanto, o encadeamento de métodos:

```
USER>do array.$push(11).$push(12).$push(13)  
USER>write array.$toJSON()  
[1,"This is a string.",{"name":"Stefan Wittmann","lastSeriesSeen":"Daredevil","likes":  
:"Galaxy"},null,null,null,null,null,null,null,"This is a string in a sparse array",11  
,12,13]
```

Integração COS apertada

Todos os exemplos de código que forneci até agora criaram os objetos dinâmicos e matrizes explicitamente. Chamamos o construtor da classe correspondente `-%Object` ou `-%Array` - e começamos a manipular o objeto na memória.

Com a nova API, há uma maneira ainda mais simples de criar objetos e matrizes dinâmicos, criando-os implicitamente com a sintaxe JSON:

```
USER>set object = {"name":"Stefan Wittmann","lastMovieSeen":"The Martian","likes":"Writing Blogs"}
USER>write object.$toJSON()
{"name":"Stefan Wittmann","lastMovieSeen":"The Martian","likes":"Writing Blogs"}
USER>set array = [1,2,3,[4,5,6],true,false,null]
USER>write array.$toJSON()
[1,2,3,[4,5,6],true,false,null]
```

Não é emocionante? É uma forma muito clara, compacta e legível de descrever o que você deseja criar. Você pode ter percebido que o array é inicializado com os valores JSON `true`, `false` e `null`. Esses valores não são diretamente acessíveis no COS, mas podem ser usados na sintaxe JSON.

Nós não paramos por aí. Para tornar isso realmente útil e dinâmico, permitimos que os valores sejam expressões COS. Considere este exemplo:

```
USER>set name = "Stefan"
USER>set subObject = {"nationality":"German","favoriteColors":["yellow","blue"]}
USER>set object = {"name":name,"details":subObject,"lastUpdate":$ZD($H,3)}
USER>write object.$toJSON()
{"name":"Stefan","details":{"nationality":"German","favoriteColors":["yellow","blue"]},
," lastUpdate ":"2016-01-31"}
```

Isso permite que você produza e altere facilmente estruturas JSON no servidor. No entanto, há uma coisa que você deve considerar: o acesso a valores sempre produzirá um valor amigável de COS. Vamos explorar um exemplo para entender o que isso realmente significa:

```
USER>set array = [1,2,3,[4,5,6],true,false,null]
USER>set iter = array.$getIterator()
USER>while iter.$getNext(.key,.value) { write "key "_key_" : "_value,! }
key 0:1
key 1:2
key 2:3
key 3:5@%Library.Array
key 4:1
key 5:0
key 6:
```

A saída até a chave 4 deve ser esperada. A chave 4 retorna o valor COS 1 para o valor JSON `true`. Semelhante para a chave 5, que retorna o valor COS 0 para o valor JSON `false`. Provavelmente menos óbvio é que a chave 6 está retornando uma string vazia para o valor JSON `null`.

A razão para isso é que queremos retornar valores amigáveis de COS que podem ser usados diretamente em condicionais de COS. Mapeando `true` para 1 e `false` para 0, você pode testar diretamente a veracidade e a falsidade em uma instrução `if`. Uma string `null` é o mais próximo que você pode chegar de um nulo JSON.

Mas como você distingue os pares de valores JSON true e 1, false e 0 e null e “ ” um do outro? Você faz isso verificando o tipo. O método que você deseja usar para isso é \$getTipoOf().

```
USER>w array.$getTipoOf(5)
boolean
USER>w array.$getTipoOf(6)
null
```

Para fechar a lacuna, você pode passar um tipo no setter para objetos dinâmicos e matrizes para especificar qual tipo o valor representa:

```
USER>do array.$set(7,1)
USER>write array.$toJSON()
[1,2,3,[4,5,6],true,false,null,1]
USER>do array.$set(7,1,"boolean")
USER>write array.$toJSON()
[1,2,3,[4,5,6],true,false,null,true]
```

Primeiro, definimos a chave 7 com o valor 1, que obviamente se traduz no número JSON 1. Se quisermos definir o valor verdadeiro, podemos especificar o tipo “booleano” para o configurador. Deixo o exercício com o valor JSON nulo para o leitor.

Por trás das cenas

Parabéns por chegar tão longe. Isso é muita informação e você ainda está lendo. Obviamente, há mais chamadas de API para aprender, mas gostaria de referir a documentação, bem como a documentação da classe para este tópico. Obviamente, existem tópicos mais avançados que podemos abordar e gostaria de abordar dois deles.

Performance

Desempenho é uma palavra difícil. Significa tantas coisas e você deve ter muito cuidado para afirmar o que quer dizer quando faz uso dele. Uma das deficiências da API antiga, que inclui o zenProxyObject, é seu comportamento de tempo de execução não linear para serializar e desserializar JSON. Embora consumir conteúdo JSON menor seja aceitável, consumir um conteúdo JSON bruto de até 100 MB no disco pode levar alguns minutos. Uma razão para isso é que a análise é implementada no COS, que não é a linguagem mais eficiente para analisar fluxos de caracteres. Além disso, o gráfico completo do objeto teve que ser construído na memória.

Tomamos muito cuidado para resolver esse problema com a nova API. A análise é implementada diretamente no kernel e inventamos uma estrutura altamente otimizada na memória para gerenciar entidades dinâmicas. A tabela a seguir mostra os resultados de um teste de desempenho que conduzimos recentemente.

Testamos o zenProxyObject no Caché 2015.1, o novo suporte JSON no Caché 2016.1 e como uma comparação de terceiros, rodamos o mesmo teste no Node.JS. Carregamos um arquivo de 10,8 MB com 1.000 empresas JSON e vários objetos incorporados e você pode ver que os números caíram de 28.000ms para 94ms com o novo suporte JSON. Melhor ainda, o novo suporte JSON é tão rápido quanto executar a mesma operação no Node. Aumentar a escala do teste torna a melhoria ainda mais clara. Consumir um arquivo com 10.000 empresas JSON melhorou de 386.700 ms para 904 ms, o que está no mesmo nível do Node novamente. As operações na memória são muito rápidas e lineares, mas, como esperado, o Node é um vencedor claro aqui, pois há um suporte nativo para objetos JSON na memória.

No geral, estamos muito felizes com os números. Conte-nos sobre sua experiência!

Métodos de Sistema

Este é o último tópico sobre o qual desejo lançar alguma luz antes de concluir. Você deve estar se perguntando por que todos os nomes de métodos começam com um caractere de dólar, \$new(), \$set(), \$pop(), \$size() e assim por diante. Esta é uma nova categoria de métodos que introduzimos, chamados métodos de sistema. Existem dois tipos diferentes de métodos de sistema, semelhantes aos métodos padrão: métodos de instância e de classe.

Os métodos do sistema não podem ser substituídos por métodos padrão, pois vivem em um namespace separado. Se você trabalhou com o zenProxyObject, provavelmente sabe que reservamos alguns nomes de propriedade. A razão é que o zenProxyObject definiu algumas propriedades e métodos internos que permitiram gerenciar seu estado e fornecer a API. Eles pisaram no namespace do usuário e tiveram que ser reservados.

Com a introdução de métodos de sistema, nos livramos desse problema e não há nomes de propriedade reservados. Seus dados JSON podem ser verdadeiramente dinâmicos de agora em diante. Os métodos do sistema são sempre prefixados com um caractere de dólar.

Conclusão

Acreditamos que o novo suporte JSON é um grande passo à frente e ajudará você a construir interfaces melhores com mais rapidez. Esperamos ver alguns benefícios em lidar com grandes conjuntos de dados conforme a API é adotada por outras partes de nossa pilha, por exemplo, DeepSee. Se você ainda não teve a chance de experimentar a nova API, experimente. Espero que você esteja tão animado quanto nós, pois essa é a base para outro recurso que irei discutir em uma postagem separada no blog em breve. Fique atento.

Estou ansioso por seus comentários.

Stefan

[#Frontend](#) [#Interoperabilidade](#) [#JSON](#) [#Node.js](#) [#ObjectScript](#) [#ZEN](#) [#Caché](#)

URL de origem: <https://pt.community.intersystems.com/post/apresentando-novos-recursos-json-no-cach%C3%A9-20161>