Le format Redbin

Table of Contents

Spécification version 1

Le format Redbin est un format binaire qui représente avec précision les valeurs Red stockées en mémoire, tout en permettant un chargement rapide (évitant l'étape d'analyse et de validation du format de représentation textuel). Le format Redbin est largement inspiré par REBin. Redbin peut encoder l'information de contrainte (binding) pour les mots et peut gérer les cycles dans les valeurs de type `any-block!`.

L'interface utilisateur d'accès au format Redbin sera fournie par `load/binary` et `mold/binary`. L'implementation sous-jacente *pourrait* utiliser le sous-système de codec, une fois qu'il sera disponible.

Contraintes d'implémentation:

- L'adresse de base en mémoire des données Redbin à charger, doit être alignée sur 64 bits.

1. Format d'encodage

Le format d'encodage *par défaut* est optimisé pour la vitesse de décodage, tandis que le format *compact* requiert un moindre espace de stockage (au prix d'un décodage beaucoup plus lent).

Les valeurs sont stockées au format little-endian (octet de poids faible en premier).

Conventions lexicales:

1. *Les nombres entre parenthèses indiquent la taille du champ en octets.*
2. *Les noms de champ suivis par un nom de type de données dans un bloc sont des espaces réservés pour une valeur de ce type de données.*
3. *Les noms de champ suivis par un signe égal ont une valeur fixée.*

2. En-tête

```
magic="REDBIN" (6), version=1 (1), flags (1), length (4), size (4)

flags (l'option est activée si le bit est mis):
    bit0: mode compact
    bit1: compressé
    bit2: table des symboles
    bit3-7: <réservés>

length : nombre d'enregistrement racines à charger.
size   : taille en octets du contenu des enregistrements.
```

Si la compression est appliquée, les données suivant l'en-tête sont le contenu devant être compressé. Le choix de l'algorithme de compression dépend de l'implémentation.

3. Table des symboles

La table des symboles suit immédiatement les données d'en-tête. Elle est optionnelle et devrait n'être utilisée que si les mots sont présents dans le reste du contenu Redbin. La table des symboles a deux sections:

- une table de distances aux représentations sous forme de chaîne de chaque symbole,
- des buffers de chaînes, terminées par NUL et concaténées les unes aux autres.

La position d'un symbole dans la table est son *index* (démarrant à zéro), et elle est utilisée comme référence pour un symbole dans les contextes et les mots. La section des buffers de chaînes contient des chaînes encodées en UTF-8 avec un remplissage optionnel à la fin pour assurer l'alignement sur 64 bits. Les distances dans la table sont des distances en octets à partir du début de la section des buffers de chaînes jusqu'au buffer de chaîne auquel il est fait référence.

Encodage de la table:

```
Défaut: length (4), size (4), offset1 (4), offset2 (4),...
Compact: TBD
```

Le champ `length` contient le nombre d'entrées dans la table. Le champ `size` indique la taille en octets du buffer de chaîne (incluant les octets optionnels de remplissage terminal).

Durant le processus de décodage, ces symboles sont fusionnés avec la table de symboles propre à Red et les distances sont remplacées par la valeur d'ID du symbole de la table Red. Cela signifie que les références aux symboles dans les enregistrements Redbin sont une référence indirecte aux entrées de la table de symboles interne à Red, qui est utilisée seulement durant le processus de chargement.

Après la Table des Symboles, les valeurs de Red sont stockées comme des enregistrements en

séquence sans délimiteur spécial ni marqueur de fin. Les valeurs chargées à partir du niveau racine sont habituellement stockées dans une série de type `block!`.

4. Définitions des enregistrements

Chaque enregistrement commence avec un champ `header` (en-tête) de 32 bits défini comme:

- bit 31 : drapeau retour-à-la-ligne
- bit 30 : drapeau pas-de-valeurs (pour les contextes)
- bit 29 : drapeau stack? (pour les contextes)
- bit 28 : drapeau self? (pour les contextes)
- bit 27 : drapeau set? (pour les mots)
- bits 26-16 : <réservé>
- bits 15-8 : unit (utilisé pour encoder la taille des éléments dans un buffer de série)
- bits 7-0 : type

Ci-dessous la description individuelle de chaque enregistrement:

4.1. Padding

```
Défaut: header (4)
Compact: n/a

header/type=0
```

Cet emplacement de type vide est utilisé pour aligner correctement les valeurs sur 64 bits.

4.2. Datatype!

```
Défaut: header (4), value (4)
Compact: TBD

header/type=1
```

4.3. Unset!

```
Défaut: header (4)
Compact: TBD

header/type=2
```

4.4. None!

```
Défaut: header (4)
Compact: TBD

header/type=3
```

4.5. Logic!

```
Défaut: header (4), value=0|1 (4)
Compact: TBD

header/type=4
```

4.6. Block!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=5
```

Le champ `head` indique la distance de la référence du bloc, en utilisant un entier démarrant à zéro. Le champ `length` contient le nombre de valeurs à stocker dans le bloc. Les valeurs du bloc suivent simplement la définition du bloc, aucun séparateur ou délimiteur de fin n'est requis.

4.7. Paren!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=6
```

Mêmes règles d'encodage que pour `block!`.

4.8. String!

```
Défaut: header (4), head (4), length (4), data (unit*length) [, padding (1-3)]
Compact: TBD

header/type=7
header/unit=1|2|4
```

Le champ `head` a la même signification que pour les blocs. Le sous-champ `unit` indique le format d'encodage de la chaîne, seules les valeurs de 1, 2 et 4 sont valides. Le champ `length` contient le nombre de points de code à stocker dans la chaîne, qui peut aller jusqu'à 16777215 (2^24 - 1) points de code. La chaîne est encodée au format UCS-1, UCS-2 ou UCS-4. Aucun caractère NUL n'est présent, ni pris en compte dans le champ `length`. Un remplissage final optionnel de 1 à 3 octets NUL peut être présent pour aligner la fin de l'enregistrement `string!` sur un groupe de 32 bits.

4.9. File!

```
Défaut: header (4), head (4), length (4), data (unit*length)
Compact: TBD

header/type=8
header/unit=1|2|4
```

Mêmes règles d'encodage que pour `string!`.

4.10. Url!

```
Défaut: header (4), head (4), length (4), data (unit*length)
Compact: TBD

header/type=9
```

Mêmes règles d'encodage que pour `string!`.

4.11. Char!

```
Défaut: header (4), value (4)
Compact: TBD

header/type=10
```

4.12. Integer!

```
Défaut: header (4), value (4)
Compact: TBD

header/type=11
```

4.13. Float!

```
Défaut: [padding=0 (4),] header (4), value (8)
Compact: TBD

header/type=12
```

Le champ optionnel de remplissage (padding) est ajouté pour aligner correctement la position du champ `value` sur une valeur de 64 bits.

4.14. Context!

```
Défaut: header (4), length (4), symbol1 (4), symbol2 (4),..., value1 [any-type!],
value2 [any-type!], ...
Compact: TBD

header/type=14
header/no-values=0|1
header/stack?=0|1
header/self?=0|1
```

Les contextes sont des valeurs Red utilisées en interne par certains types de données comme `function!`, `object!` et les types dérivés. Un contexte contient deux tables consécutives, la première est la liste des mots du contexte représentés comme des références symboliques, la seconde est celle des valeurs associées à chacun des symboles de la première table. Le champ `length` indique le nombre d'entrées dans le contexte. Les enregistrements de contexte ne peuvent exister qu'au niveau racine, il ne peuvent être imbriqués. Si le drapeau `no-values` est mis, cela signifie qu'il n'y a pas de valeurs suivant les symboles (contenu vide). Si le drapeau `stack` est mis, alors les valeurs sont allouées sur la pile et non sur le tas de la mémoire. Le drapeau `self?` est utilisé pour indiquer que le contexte est capable de gérer un mot auto-référençant (`self` dans des objets).

4.15. Word!

```
Défaut: header (4), symbol (4), context (4), index (4)
Compact: TBD

header/type=15
header/set?=0|1
```

Le champ `context` est une position relative au début de la section des enregistrements dans le fichier Redbin, faisant référence à une valeur de type `context!`. Le contexte doit être localisé avant l'enregistrement du mot dans la liste des enregistrements Redbin. Si `context` est égal à `-1`, cela fait référence au contexte global.

Si le champ `field` est défini, cet enregistrement est suivi par un enregistrement de type `any-value!`, et cette valeur devra être donnée au mot (dans le bon contexte) par le décodeur. Cela forme un couple nom/valeur permettant d'encoder des valeurs de mots de manière ad hoc, lorsqu'il serait trop coûteux de fournir une séquence de valeurs pour un contexte donné (essentiellement pour des couples nom/valeur dans le contexte global).

4.16. Set-word!

```
Défaut: header (4), symbol (4), context (4), index (4)
Compact: TBD

header/type=16
```

Même chose que pour `word!`.

4.17. Lit-word!

```
Défaut: header (4), symbol (4), context (4), index (4)
Compact: TBD

header/type=17
```

Même chose que pour `word!`.

4.18. Get-word!

```
Défaut: header (4), symbol (4), context (4), index (4)
Compact: TBD

header/type=18
```

Même chose que pour `word!`.

4.19. Refinement!

```
Défaut: header (4), symbol (4), context (4), index (4)
Compact: TBD

header/type=19
```

Même chose que pour `word!`.

4.20. Issue!

```
Défaut: header (4), symbol (4)
Compact: TBD

header/type=20
```

4.21. Native!

```
Défaut: header (4), ID (4), spec [block!]
Compact: TBD

header/type=21
```

`ID` est une position relative dans la table de sauts `natives/table`.

4.22. Action!

```
Défaut: header (4), ID (4), spec [block!]
Compact: TBD

header/type=22
```

`ID` est une position relative dans la table de sauts `actions/table`.

4.23. Op!

```
Défaut: header (4), symbol (4),
Compact: TBD

header/type=23
```

`symbol` représente le nom de l'action, de la native ou de la fonction (du contexte global seulement) utilisé(e) comme source pour cette valeur d' `op!`.

4.24. Function!

```
Défaut: header (4), context [context!], spec [block!], body [block!], args [block!],
obj-ctx [context!]
Compact: TBD

header/type=24
```

4.25. Path!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=25
```

Mêmes règles d'encodage que pour `block!`.

4.26. Lit-path!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=26
```

Mêmes règles d'encodage que pour `block!`.

4.27. Set-path!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=27
```

Mêmes règles d'encodage que pour `block!`.

4.28. Get-path!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=28
```

Mêmes règles d'encodage que pour `block!`.

4.29. Bitset!

```
Défaut: header (4), length (4), bits (length)
Compact: TBD

header/type=30
```

Les champs `length` indiquent le nombre de bits stockés, arrondi au multiple de 8 supérieur. Les bits sont des copies mémoire du buffer de série `bitset!`. L'ordre des octets est préservé. Le champ `bits` doit être complété avec suffisamment d'octets NUL pour conserver l'alignement à 32 bits de l'enregistrement suivant.

4.30. Point!

```
Défaut: header (4), x (4), y (4), z (4)
Compact: TBD

header/type=31
```

4.31. Object!

```
Défaut: header (4), context [reference!], class-id (4), on-set-idx (4), on-set-arity
(4)
Compact: TBD

header/type=32
```

Le champ `on-set-idx` indique la position relative de `on-change*` dans la table des valeurs du contexte. `on-set-arity` stocke l'arité de cette fonction.

4.32. Typeset!

```
Défaut: header (4), array1 (4), array2 (4), array3 (4)
Compact: TBD

header/type=33
```

4.33. Error!

```
Défaut: header (4), context [reference!]
Compact: TBD

header/type=34
```

4.34. Vector!

```
Défaut: header (4), head (4), length (4), values (unit*length)
Compact: TBD

header/type=35
```

`unit` indique la taille du type d'élément vectoriel: 1, 2, 4 ou 8 octets. Le champ `values` contient la liste des valeurs. `values` doit être complété avec des octets NUL pour aligner à 32 bits l'enregistrement suivant (si `unit` est égal à 1 ou 2).

4.35. Pair!

```
Défaut: header (4), x (4), y (4)
Compact: TBD

header/type=37
```

4.36. Percent!

```
Défaut: [padding=0 (4),] header (4), value (8)
Compact: TBD

header/type=38
```

La valeur de pourcentage est stockée comme un nombre à virgule flottante sur 64 bits. Le champ de remplissage optionnel est ajouté pour aligner correctement la position du champ `value` sur une valeur de 64 bits.

4.37. Tuple!

```
Défaut: header (4), array1 (4), array2 (4), array3 (4)
Compact: TBD

header/type=39
```

4.38. Map!

```
Défaut: header (4), length (4), ...
Compact: TBD

header/type=40
```

Le champ `length` contient le nombre d'éléments (clés + valeurs) à stocker dans la table (map). Les éléments de la table suivent simplement l'indication de la longueur, aucun séparateur ou délimiteur de fin n'est requis.

4.39. Binary!

```
Défaut: header (4), head (4), length (4), ...
Compact: TBD

header/type=41
```

Mêmes règles d'encodage que pour `block!`.

4.40. Time!

```
Défaut: [padding=0 (4),] header (4), value (8)
Compact: TBD

header/type=43
```

La valeur d'heure est stockée au format 64 bits. Le champ de remplissage optionnel est ajouté pour aligner correctement la position du champ `value` sur une valeur de 64 bits.

4.41. Tag!

```
Défaut: header (4), head (4), length (4), data (unit*length)
Compact: TBD

header/type=44
header/unit=1|2|4
```

Mêmes règles d'encodage que pour `string!`.

4.42. Email!

```
Défaut: header (4), head (4), length (4), data (unit*length)
Compact: TBD

header/type=45
header/unit=1|2|4
```

Mêmes règles d'encodage que pour `string!`.

4.43. Date!

```
Défaut: header (4), date (4), time (8)
Compact: TBD

header/type=47
```

La date est codée dans un entier de 32 bits (comme dans `red-date!`). La valeur d'heure est stockée au format 64 bits.

4.44. Reference!

```
Défaut: header (4), count (4), index1 (4), index2 (4), ...
Compact: TBD

header/type=255
```

Ce type spécial d'enregistrement stocke une référence à une valeur déjà chargée de type `any-block!` ou `object!`. Cela rend possible de stocker des cycles en Redbin. La référence est créée à partir d'un chemin vers les valeurs chargées (en supposant que les valeurs racine sont stockées dans un bloc). Chaque champ `index` pointe vers la valeur de série ou d'objet dans laquelle aller, jusqu'à ce que la dernière soit atteinte, pointant vers la valeur à laquelle il doit être fait référence. Le champ `count` indique le nombre d'index à traverser. Si l'un des index doit être appliqué à un objet, il fait référence au champ correspondant de l'objet (0 ⇒ premier champ, 1 ⇒ 2e champ,...). Tous les index démarrent à zéro.