Redbin フォーマット

目次

1. エンコーディングフォーマット . . . 2
2. ヘッダー . . . 2
3. シンボルテーブル . . . 3
4. レコード定義 . . . 3
    4.1. パディング . . . 4
    4.2. データ型 . . . 4
    4.3. Unset! . . . 4
    4.4. None! . . . 4
    4.5. Logic! . . . 5
    4.6. Block! . . . 5
    4.7. Paren! . . . 5
    4.8. String! . . . 5
    4.9. File! . . . 6
    4.10. Url! . . . 6
    4.11. Char! . . . 6
    4.12. Integer! . . . 6
    4.13. Float! . . . 6
    4.14. Context! . . . 7
    4.15. Word! . . . 7
    4.16. Set-word! . . . 7
    4.17. Lit-word! . . . 8
    4.18. Get-word! . . . 8
    4.19. Refinement! . . . 8
    4.20. Issue! . . . 8
    4.21. Native! . . . 9
    4.22. Action! . . . 9
    4.23. Op! . . . 9
    4.24. Function! . . . 9
    4.25. Path! . . . 9
    4.26. Lit-path! . . . 10
    4.27. Set-path! . . . 10
    4.28. Get-path! . . . 10
    4.29. Bitset! . . . 10
    4.30. Point! . . . 11
    4.31. Object! . . . 11
    4.32. Typeset! . . . 11

4.33. Error! 11
4.34. Vector! 11
4.35. Pair! 12
4.36. Percent! 12
4.37. Tuple! 12
4.38. Map! 12
4.39. Binary! 12
4.40. Time! 13
4.41. Tag! 13
4.42. Email! 13
4.43. Date! 13
4.44. Reference! 13

仕様バージョン *1*

Redbinはメモリ中のRedの値を正確に保存するためのバイナリフォーマットで、（テキスト表現のフォーマットの場合に必要なパースや値の検証をスキップすることで）高速な読み込みを可能にします。RedbinフォーマットはREBinに大きくインスパイアされています。Redbinはwordへのバインディング情報もエンコードでき、any-block!型における再帰的なデータも扱えます。

Redbinフォーマットにアクセスするためのユーザーインターフェイスは`load/binary`と`mold/binary`によって提供される予定です。将来的にコーデックサブシステムが使用可能になることで、ベースとなる実装がそれを使用するかもしれません。

実装上の制約:

- ロードされるRedbinデータのメモリ上のベーすアドレスは64ビットアラインメントである必要があります。

1. エンコーディングフォーマット

デフォルトのエンコーディングフォーマットはデコードのスピードに最適化されています。一方で比較的小さなストレージ空間では（デコーディングスピートを犠牲にして）より小さなコンパクトなフォーマット が必要とされることもあります。

値はリトルエンディアンフォーマットで保存されます。

構文規則:

1. 丸括弧の中の数字はフィールドのバイトサイズを示します。
2. ブロック内のデータタイプ名の後のフィールド名はそのデータタイプの値に対するプレースホルダです。
3. イコールの後のフィールド名は固定値です。

2. ヘッダー

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

flags (ビットがセットされているとオプションが有効になります):
    bit0: コンパクトモード
    bit1: 圧縮モード
    bit2: シンボルテーブル
    bit3-7: <将来のために予約>

length : ロードするルートレコードの数
size   : バイトの中のレコードペイロードのサイズ
```

圧縮が行われている場合、ヘッダーの後に続くデータは圧縮されたペイロードになります。圧縮アルゴリズムの選択は実装に依存します。

3. シンボルテーブル

シンボルテーブルはヘッダーデータの直後に続きます。それはオプショナルで残りのRedbinペイロードにwordが存在する場合にだけ使われるべきです。シンボルテーブルは次の２つのセクションを持ちます。

- それぞれのシンボルの文字列表現へのオフセットのテーブル
- NUL終端でお互いに連結された文字列バッファ

テーブルの中のシンボルの位置はゼロベースの*index*です。それはコンテキストとwordのシンボルへの参照として使われます。文字列バッファのセクションは64ビットアラインメントを保証するために終端までパディングされたUTF-8エンコードの文字列を含みます。テーブルの中のオフセットは文字列バッファセクションの文字列の始まりから参照される文字列バッファまでのバイトのオフセットです。

テーブルのエンコーディング：

```
デフォルト: length (4), size (4), offset1 (4), offset2 (4),...
コンパクト: (確認中)
```

`length` フィールドはテーブルの中の数々のエントリーを含んでいます。 `size` フィールドは（オプショナルな終端のパディングバイトを含む）バイト中の文字列バッファのサイズを示します。

デコーディングプロセスの間、これらのシンボルはRed自身のシンボルテーブルにマージされ、オフセットはRedテーブルのシンボルIDの値で置き換えられます。すなわち、Redbinレコード内のシンボル参照はローディングプロセスの間しか使用されず、Redの内部的なシンボルテーブルエントリへの間接的な参照となります。

シンボルテーブルの後には、Redの値が特別なデリメタやエンドマーカーはなしで連続したれコードとして保持されています。ルートレベルからロードされた値は通常 block! series として保持されます。

4. レコード定義

各レコードは以下のような32ビットの `header` フィールドで始まります。

- bit31 : new-line（改行）フラグ
- bit30 : no-values（値なし）フラグ（contextのため）
- bit29 : stack?（スタック）フラグ（contextのため）
- bit28 : self?（セルフ）フラグ（コンテキストのため）
- bit27 : set?（セット）フラグ（wordのため）
- bit26-16 : <将来のために予約>
- bit15-8 : ユニット（seriesのバッファにおいてエンコーディングする要素のサイズのため）
- bit7-0 : タイプ

各レコードの説明は以下の通りです。

4.1. パディング

```
デフォルト: header (4)
コンパクト: n/a

header/type=0
```

この空のタイプのスロットは値を適切に64ビットにアラインメントするために使われます。

4.2. データ型

```
デフォルト: header (4), value (4)
コンパクト: 未定

header/type=1
```

4.3. Unset!

```
デフォルト: header (4)
コンパクト: 未定

header/type=2
```

4.4. None!

```
デフォルト: header (4)
コンパクト: 未定

header/type=3
```

4.5. Logic!

```
デフォルト: header (4), value=0|1 (4)
コンパクト: 未定

header/type=4
```

4.6. Block!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=5
```

`head` フィールドはブロックの参照へのオフセットを示すゼロベースの整数値です。 `length` フィールドはブロックの中に保持されている値の数を保持します。ブロックの値は単純にブロックの定義の後に続けられ、セパレータや区切り文字は必要ありません。

4.7. Paren!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=6
```

block!と同様のエンコーディングルールです。

4.8. String!

```
デフォルト: header (4), head (4), length (4), data (unit*length) [, padding (1-3)]
コンパクト: 未定

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

`head` フィールドはブロックと同様の意味を持ちます。 `unit` サブフィールドは文字列のエンコーディングフォーマットを示します。値として有効なのは1、2、4のみです。 `length` フィールドは保存されている文字列のコードポイントの番号を含んでいます。16777215(2^24 - 1)までのコードポイントがサポートされます。文字列はUCS-1、UCS-

2またはUCS-4のフォーマットでエンコードされます。NUL文字は存在せず、フィールドにもカウントされません。必要な場合、末尾のパディングが1から3のNULバイトで行われ、32ビットを境界とした文字列レコードのアラインの終端を示します。

`length`

4.9. File!

```
デフォルト: header (4), head (4), length (4), data (unit*length)
コンパクト: 未定

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

文字列と同様のエンコーディングルールです。

4.10. Url!

```
デフォルト: header (4), head (4), length (4), data (unit*length)
コンパクト: 未定

header/type=9
```

文字列と同様のエンコーディングルールです。

4.11. Char!

```
デフォルト: header (4), value (4)
コンパクト: 未定

header/type=10
```

4.12. Integer!

```
デフォルト: header (4), value (4)
コンパクト: 未定

header/type=11
```

4.13. Float!

```
デフォルト: [padding=0 (4),] header (4), value (8)
コンパクト: 未定

header/type=12
```

`value` フィールドを64ビットの値に適切にアラインするために、必要なパディングフィールドが追加されます。

4.14. Context!

```
デフォルト: header (4), length (4), symbol1 (4), symbol2 (4),..., value1 [any-type!], value2
[any-type!], ...
コンパクト: 未定

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

コンテキストはfunction!、object!および派生する型において内部的に使用されるRedの値です。コンテキストは連続的な２つのテーブルを含みます。１つはシンボルの参照として表現されるコンテキストの中のwordエントリのリストです。もう１つが１つ目のテーブルのシンボルそれぞれに対して関連付けされる値です。 `length` フィールドはコンテキスト内のエントリの数を示します。コンテキストレコードはルートレベルでのみ存在でき、ネストすることはできません。もし `no-values` フラグがセットされている場合、続くシンボルには値がない（空のコンテキストである）ことを示します。もし `stack?` フラグがセットされている場合、値はヒープではなくスタックに確保されます。 `self?` フラグはコンテキストが自分自身を参照するword（オブジェクト内の `self` ）をハンドルできるかを示すために使われます。

4.15. Word!

```
デフォルト: header (4), symbol (4), context (4), index (4)
コンパクト: 未定

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

`context` フィールドはcontext!型の値を示すRedbinファイル内のレコードセクションの開始位置からのオフセットです。コンテキストはRedbinのレコードリスト内のwordのレコードの前に位置している必要があります。 `context` が -1 の場合はグローバルコンテキストになります。

`set?` フィールドが定義されている場合、このレコードの後に [any-value!] 型のレコードが続き、デコーダにより正しいコンテキスト内のwordにその値がセットされる必要があります。これは指定のコンテキストへ一連の値を与えるコストが大きすぎる（主にグローバルコンテキストの名前と値のペアを渡すような）場合に、wordの値をアドホックにエンコードするための名前と値のペアを形成します。

4.16. Set-word!

```
ﾃﾞﾌｫﾙﾄ:  header (4), symbol (4), context (4), index (4)
コンパクト: 未定

header/type=16
```

word!と同様です。

4.17. Lit-word!

```
ﾃﾞﾌｫﾙﾄ:  header (4), symbol (4), context (4), index (4)
コンパクト: 未定

header/type=17
```

word!と同様です。

4.18. Get-word!

```
ﾃﾞﾌｫﾙﾄ:  header (4), symbol (4), context (4), index (4)
コンパクト: 未定

header/type=18
```

word!と同様です。

4.19. Refinement!

```
ﾃﾞﾌｫﾙﾄ:  header (4), symbol (4), context (4), index (4)
コンパクト: 未定

header/type=19
```

word!と同様です。

4.20. Issue!

```
ﾃﾞﾌｫﾙﾄ:  header (4), symbol (4)
コンパクト: 未定

header/type=20
```

4.21. Native!

```
デフォルト: header (4), ID (4), spec [block!]
コンパクト: 未定

header/type=21
```

`ID` は内部の `natives/table` へのジャンプテーブルへのオフセットです。

4.22. Action!

```
デフォルト: header (4), ID (4), spec [block!]
コンパクト: 未定

header/type=22
```

`ID` は内部の `actions/table` へのジャンプテーブルへのオフセットです。

4.23. Op!

```
デフォルト: header (4), symbol (4),
コンパクト: TBD

header/type=23
```

`symbol` はそのop!型の値の元となったaction、nativeまたはfunctionの名前（グローバルコンテキストで定義されたものに限ります）を表します。

4.24. Function!

```
デフォルト: header (4), context [context!], spec [block!], body [block!], args [block!], obj-
ctx [context!]
コンパクト: 未定

header/type=24
```

4.25. Path!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=25
```

block!と同じエンコーディングルールです。

4.26. Lit-path!

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

header/type=26
```

block!と同じエンコーディングルールです。

4.27. Set-path!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=27
```

block!と同じエンコーディングルールです。

4.28. Get-path!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=28
```

block!と同じエンコーディングルールです。

4.29. Bitset!

```
デフォルト: header (4), length (4), bits (length)
コンパクト: 未定

header/type=30
```

`length`
フィールドは保持されているビットの数を示しています。ビットは8の倍数で切り上げて丸められます。ビットはbitset!のseriesバッファのメモリダンプになります。バイトの順番は保持されます。
`bits` フィールドは次のレコードが32ビットアラインになるようにNULバイトで埋められる必要があります。

4.30. Point!

```
デフォルト: header (4), x (4), y (4), z (4)
コンパクト: 未定

header/type=31
```

4.31. Object!

```
デフォルト: header (4), context [reference!], class-id (4), on-set-idx (4), on-set-arity (4)
コンパクト: 未定

header/type=32
```

`on-set-idx` フィールドはコンテキストの値テーブル内の `on-change*` へのオフセットを示します。 `on-set-arity` はその関数へのアリティを保持します。

4.32. Typeset!

```
デフォルト: header (4), array1 (4), array2 (4), array3 (4)
コンパクト: 未定

header/type=33
```

4.33. Error!

```
デフォルト: header (4), context [reference!]
コンパクト: 未定

header/type=34
```

4.34. Vector!

```
デフォルト: header (4), head (4), length (4), values (unit*length)
コンパクト: 未定

header/type=35
```

`unit` はvectorエレメントのタイプサイズの種類を示し、1、2、4、8バイトになります。 `values` フィールドは値のリストを保持します。 `values` は（もし `unit` が1か2の場合、）次のレコードの境界が32ビットになるように、NULバイトで埋められる必要があります。

4.35. Pair!

```
デフォルト: header (4), x (4), y (4)
コンパクト: 未定

header/type=37
```

4.36. Percent!

```
デフォルト: [padding=0 (4),] header (4), value (8)
コンパクト: 未定

header/type=38
```

percent!の値は64ビット浮動小数点で保持されます。`value` フィールドのオフセットが64ビット値になるようにアラインするため、オプショナルなパディングフィールドが追加されます。

4.37. Tuple!

```
デフォルト: header (4), array1 (4), array2 (4), array3 (4)
コンパクト: 未定

header/type=39
```

4.38. Map!

```
デフォルト: header (4), length (4), ...
コンパクト: 未定

header/type=40
```

`length` フィールドはmap内に保持される（キーと値を合わせた）要素の数です。mapの要素はlengthで示される終わりまで単純に連続し、セパレータやデリメタは必要ありません。

4.39. Binary!

```
デフォルト: header (4), head (4), length (4), ...
コンパクト: 未定

header/type=41
```

block!と同じエンコーディングです。

4.40. Time!

```
デフォルト: [padding=0 (4),] header (4), value (8)
コンパクト: 未定

header/type=43
```

time!の値は64ビット浮動小数点数で保持されます。 `value` フィールドのオフセットが64ビット値になるようにアラインするため、オプショナルなパディングフィールドが追加されます。

4.41. Tag!

```
デフォルト: header (4), head (4), length (4), data (unit*length)
コンパクト: 未定

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

string!と同じエンコーディングルールです。

4.42. Email!

```
デフォルト: header (4), head (4), length (4), data (unit*length)
コンパクト: 未定

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

string!と同じエンコーディングルールです。

4.43. Date!

```
デフォルト: header (4), date (4), time (8)
コンパクト: 未定

header/type=47
```

dateは（ `red-date!` と同様）32ビットintegerでパックされます。timeは64ビット浮動小数点数で保持されます。

4.44. Reference!

```
デフォルト: header (4), count (4), index1 (4), index2 (4), ...
コンパクト: 未定

 header/type=255
```

この特別なレコードタイプはany-block!またはobject!のすでにロードされた値に対する参照を保持します。これはRedbin内での循環参照を可能にします。参照は（ルートの値がblock内に保持されていると仮定して）パスからロードされた値への参照として作成されます。それぞれの `index` フィールドは最後の値になるまで、内部を参照するべきseriesかobjectを示します。 `count` フィールドは内部参照するインデックスの数を示します。インデックスの１つがobjectを指している場合、それはobjectの対応するフィールド（0の場合は1つ目のフィールド、1の場合は2つ目フィールド・・・と続きます）を参照します。全てのインデックスはゼロベースです。