「Life with JSON」Background 編

2026-01-03

じぇいそにっく・らぼ

目次

1 データモデリング 2

2 1 枚の図 2

3 構造体 2

4 ER 図 (Entity Relationship Diagram) 3

5 クラス図 3

6 JSON によるデータ表現 4

7 JSON データモデル表現 4

7.1 JSON の型 5

7.2 構造化型 5

7.3 エンティティとリレーションシップ 6

7.4 Graph 6

7.5 関係（include/ref） 6

7.6 単一継承（subtype） 6

7.7 「1 枚の図」（データモデル）の価値 7

7.8 非正規化表現（projection） 7

7.9 Graph Script 7

8 その他 8

8.1 要件はどのようにしてまとめるのか 8
8.2 JSON Schema を出力しないのか 8
8.3 JSON データが大きくなりすぎたらどうするか 9

9 Enjoy Data Modeling 9

1 データモデリング

システムを表現する上で、最小限、必要になるのは「機能」ではなく「データ」だと私は考えている。現地調整のために出張する際、できるだけ荷物を減らしたくて、「紙一枚でシステム全体を可視化できないか」と試行錯誤した。複数のデータ定義表を縮小コピーし、切り貼りして A3 一枚にまとめた。最後にそれをコピーして、切り貼りが剥がれない“完成図”を作った。当時はオフィス製品どころか Windows すら存在せず、システム設計資料はすべて紙と手書きで作成していた時代である。

2 1 枚の図

苦労して作成したその一枚の図は、システム全体を俯瞰するのに大いに役立った。もちろん、すべての情報を載せることはできず、細部は端折らざるを得なかったし、項目名が読めない箇所もあった。それでも、その図だけでシステムの構成を把握するには十分だった。細かな情報が必要な場合は、該当資料への参照を書き添えることで補った。FORTRAN やアセンブリ言語しかなかった時代である。この図は非常に有用だったが、参照資料に改訂が入るたびに作り直すのは大変で、最終的には図に手書きで修正を加えるしかなかった。

3 構造体

その後、C 言語と Unix が入ってきて、システム開発環境は大きく変わった。端末から直接ソースコードを書けるようになり、データ宣言をヘッダーファイルとして記述できるようになった。構造体を使ってシステムで扱うデータを定義できるようになると、機能（コード）とデータ型定義を分離することが可能になった。ヘッダーファイルを見ればシステムの構成が把握できるため、しばらくはこれを「1 枚の図」の代わりとして使っていた。しかし、新しいプログラミング言語が登場するたびに記述方法が変わり、結局「1 枚

の図」の代わりにはならなかった。

4　ER 図 (Entity Relationship Diagram)

リレーショナルデータベース（RDB）が普及すると、システムとデータを保存・管理する仕組みを分離できるようになった。機能とデータを切り離せるようになったことで、設計は格段に整理され、見通しが良くなった。「1 枚の図」が表していたものは、もともと“データそのもの”であり、機能とは直接関係がなかった。その意味で、RDB と ER 図の登場は、自分が長年求めていた方向性と一致していた。

ほどなく、RDB の設計には ER 図を使ってデータモデルを表現し、それをもとにデータベースを構築する方法が一般化した。この頃から私は、「1 枚の図」で表現しようとしていたものが、実は「データモデル」そのものだったのだと明確に意識するようになった。

データベース設計手法がしっかりしていれば、ER 図作成ツールはシステム構築・拡張・保守に非常に有効である。しかし、当時の既存ツールでは実施していた手法が Fit せず、ついには自作で ER 図ツールを作ってしまうほど、データモデリングにのめり込んでいった。

特に、既存システムを拡張しながら再構築する場面では、ER 図は非常に強力な武器になった。

ただし、ER 図にはいくつか気になる点もあった。

- コード体系が先に必要になる（新規作成システムでは未確定のことが多い）
- データ項目ベースで記述する必要がある。エンティティ・リレーションシップというより、エンティティ・データ項目で表現する。
- リレーションシップは項目名で表現するしかない
- RDB は表（テーブル）でしかデータを表現できず柔軟性に限界がある

エンティティが増え、項目数が増えるほど ER 図は巨大化し、A3 に収めるための図のレイアウト調整に多くの時間を取られるようになった。

5　クラス図

オブジェクト指向パラダイムが広まると、クラス図（オブジェクト図）が ER 図の代わりとして使われるようになった。クラス図は C++、Java、Object Pascal などの class 型を表現するための図であり、確かにデータも扱える。

しかし、メソッドという“機能”まで同じ図に含まれるため、私はクラス図をデータモデリングツールとして使うことには否定的である。

ただし、クラス図でひとつだけ感心した点がある。クラス間のリレーションシップが、“どのフィールドがその関係を担っているか”を明示的にマッピングできることだ。

6　JSON によるデータ表現

RDB では、エンティティはすべてテーブルで表現される。これは非常に大きな制約である。正規化されたデータベースは構築方法によっては強力だが、データを利用する側から見ると柔軟性に欠ける場面が多かった。一部の RDB では、配列型や BLOB に XML 形式のテキストを格納するなど、“異端的な”拡張が行われていたこともある。しかし、そのようなデータベースでどのようにデータモデルを作るつもりなのか、あるいはデータモデルなしで運用・保守を続けるつもりなのか、当時の利用者に問いただしたくなる場面もあった。RDB ではデータの受け渡しに CSV がよく使われる。しかし CSV は単純すぎて、「1 枚の図」で表現していたような 1:N の構造を持つデータを表すには力不足だった。私は、システムを構成するデータを十分に表現できる汎用的なデータフォーマットを求めていた。その答えが JSON（JavaScript Object Notation）だった。JSON は JavaScript との親和性が高く、Web アプリケーションの通信形式としても効率が良い。さらに、フロント側のデータ保存形式としても自然に使える。データベースの世界でも、RDB 一辺倒から NoSQL へと広がり、特にドキュメント型データベース（MongoDB など）が JSON 形式のデータを扱うようになったことで、JSON への注目は一気に高まった。ドキュメント型 DB が必ずしも JSON を必須とするわけではないが、私は JSON 形式を扱えるデータベースを総称して「JSON 系データベース」と呼んでいる。DB から JSON 形式でデータを送受信できることは、フロント側にとって余計な変換処理が不要になるため、非常に都合が良い。また JSON は JavaScript だけでなく、Java、PHP、Ruby、Python など多くの Web 系言語で扱えるため、言語に依存しないデータ形式としても優れている。そして私は、「自分が本当に実現したかったのは、JSON データをモデリングすることだ」と、この頃ようやくはっきり認識した。

7　JSON データモデル表現

JSON データをモデリングするにあたり、まず JSON の基本を簡単に振り返る。そのうえで、JSON データを表現するためのデータモデルの要件と、モデリング方針について

考えていきたい。

7.1 JSON の型

JSON には、文字列・数値・ブール値・ヌル値・オブジェクト・配列の 6 種類のデータ型がある。このうち、文字列・数値・ブール値・ヌル値はプリミティブ型で、これ以上分解できない。オブジェクトと配列は構造化型であり、JSON の大きな特徴を形作っている。数値型については注意が必要だ。JavaScript では整数と浮動小数点の区別がなく、すべて 64 ビット浮動小数点数（IEEE 754）として扱われる。整数かどうかは isInteger() によって判定されるが、「元の値」と「小数点以下を切り捨てた値」が等しい場合に整数とみなすという、やや緩い基準である。この仕様が問題になるのは、MongoDB などで JSON（正確には BSON）がバイナリ形式で保存される場合だ。静的型付け言語（Java など）で扱う際に、整数として扱われたり浮動小数点として扱われたりすることがある。致命的な問題ではないが、知っておくべき性質である。ただし、JSON が言語を超えて構造化データを扱えるメリットに比べれば、些細な問題だと考えている。データモデリングでは、整数か浮動小数点かを明確に区別する必要があるため、私は integer と number を使い分ける方針にしている。

7.2 構造化型

JSON の強みは、オブジェクトと配列という構造化型を扱える点にある。RDB のテーブルでは通常これらを直接表現できない。

7.2.1 object 型

名前／値のペアを 0 個以上持つ。値には構造化型も指定できるため、入れ子構造を自由に作れる。なお、オブジェクト内の項目順序は保証されない。

7.2.2 array 型

値の並びを保持し、順序が保証される。1 つの配列内で同じ型である必要はなく、構造化型も含められるため、順序を持つ入れ子構造を表現できる。

7.3 エンティティとリレーションシップ

データモデリングにおいて、エンティティとは「管理対象」を表す概念である。属性名／値の組み合わせで表現でき、JSON ではオブジェクト型に相当する。特定のデータを識別する属性を identifier 属性と呼ぶ。これは必ずしもテーブルやコレクションの一意キーとは限らない。特定の範囲で他と区別できれば identifier として扱える。リレーションシップは、エンティティ間の数による対応関係を表し、1:1、1:N、N:M がある。RDB の第 3 正規化モデルでは、JOIN の都合上、対応表・対照表といった中間エンティティが必要になることがある。一方 JSON では、関連データを構造化型として 1 つの JSON に含められるため、1:1 と 1:N があれば十分表現できる。JSON 系 DB は JOIN が得意ではないため、参照データを一部取り込んで JSON に含めることが多い。これは、マスターデータの改訂が過去データに影響する RDB とは異なり、「参照時点の値を保持できる」という利点にもなる。

7.4 Graph

アプリケーションで扱うデータは入れ子構造で表現されるが、マスターデータ参照まで含めると、単純な木構造では表せない。ER 図では対照表・対応表で表現していたが、JSON データの関係性を考えると、より自由度の高い表現が必要になる。また、エンティティ間の関係には方向性があり、どちらにデータを含めるか、どちらが参照するかが決まる。そこで私は、データモデルを 有向グラフ（Digraph）として表現することにした。

7.5 関係（include/ref）

エンティティ間の関係には、主に include と ref がある。- include : 一方が他方を含む- ref : 一方が他方を参照する（JSON をまたぐことが多い）JSON 系 DB ではサイズ制限があるため、意味的には include でも、実装上 ref に切り替えることがある。また、生成タイミングやライフサイクルの違いから ref を選ぶ場合もある。

7.6 単一継承（subtype）

数独の例で言えば : - ヒントセル- 入力セルどちらも「セル」だが、役割が異なる。OOP の継承に近いが、JSON は継承を表現できないため、親エンティティに discriminator

フィールドを持たせ、子エンティティの実装名を値として区別する方式を採用した。

7.7 「1 枚の図」（データモデル）の価値

データモデルはシステム開発のためだけにあるわけではない。整備されたデータモデルは、社内データの意味や関連を視覚的に理解しやすくし、経営層・現場担当者の洞察を助ける。表記揺れや重複は誤った分析を招くが、データモデルによって「顧客」「売上」などの定義が統一され、データの信頼性が向上する。データモデルは社内資産であり、システム開発だけに使うのはもったいない。AI 利用においても、データの質が性能を左右するため、構造の共有は非常に重要である。

7.8 非正規化表現（projection）

ref 関係を JSON に反映する際、関係元に関係先の属性を展開する必要がある。JSON は JOIN が弱いため、関係先のデータを埋め込む（非正規化）実装が自然になる。しかし、マスターデータには不要な属性も含まれるため、必要な属性だけを選択して取り込む仕組みが必要だった。そこで projection（属性の射影機能）を導入し、以下の 3 種類のエッジデータ型で対応した。

projection 対応データ型:

```
object<>
array<>
*object<>
```

object<> と array<> は、参照先の属性から必要なものだけを選び、新しい object を生成する。属性名を変更する（例：id → sudoku_id）こともできる。*object<> は、参照先の属性を参照元の属性として直接取り込む方式で、JSON の階層を増やさずに参照元を明確にできる。ER 図では項目名でしか参照関係を表せず、対応関係が明示できないという不満があったが、projection によってこの問題を解消できた。

7.9 Graph Script

Graph Script は、システム記述からエンティティを node として抽出し、関係を edge として表現するためのスクリプトである。モデリング初期段階の考察を整理・保存する目的で作成した。ツールに踏み込めないユーザーでも、まず Graph Script を書くこと

で一歩を踏み出せるようにした。生成されたスクリプトは json-modeler に取り込まれ、Graph として可視化される。まずは Graph をレイアウトし直すところから始めてほしい。Graph の可視化には Cytoscape.js を利用しており、ER 図ツールを自作したときに 1〜2 ヶ月かかった労力を大幅に節約できた。拡張性も高く、JSON Graph モデルの構築が非常に容易になった。Cytoscape.js とその開発者の方々には、心から感謝している。

8 その他

8.1 要件はどのようにしてまとめるのか

モデリングを行うには、まずシステムに対する要件の記述が必要になる。しかし、この「要件のまとめ方」について、私は強く言いたいことがある。要件は、1 人の考えだけでまとめるのが良いとは限らない。どれほど優秀な人であっても、すべての要求を把握することは難しい。仮に 1 人が作成した資料をもとに複数人が意見を述べ、それを 1 人がまとめたとしても、最初に作成した人の思考に引きずられる可能性が高い。また、最終的に資料をまとめるのが 1 人であることも問題だ。まとめる人がどのように意見を解釈したかによって、内容が大きく左右されてしまう。資料を多くの人に送付し、コメントをもらう方法もあるが、その場合でも「基本方針」まで踏み込んだ修正は難しい。だからこそ、関係者を集めて、皆で要件をまとめることが最も良いと私は考えている。このテーマは深いため、ここでは詳細を述べず、別途資料としてまとめる予定である。

8.2 JSON Schema を出力しないのか

当初、json-modeler では JSON Schema を出力することも検討していた。JSON Schema を調べる中で、数値型には number だけでなく integer があることも理解した。JSON Schema の有用性は、わかったが、それでも JSON Schema 出力機能を実装しなかった。理由は 2 つある。

理由 1. JSON Schema を理解できる人が限られる

JSON Schema は JSON 形式でデータ構造を定義でき、プログラムによる型整合性チェックも可能だ。しかし、これを正しく理解して作成するには習熟が必要であり、データを理解するための、データモデルを共有するという目的には必ずしも向いていない。json-modeler の目的は、「どのような JSON をシステムで使うべきかを明示すること」である。その目的のために、利用者が JSON Schema を学ぶ必要があるのは本末転倒だと感じた。

理由 2. JSON Schema のバージョン問題
JSON Schema には複数のバージョンが存在し、推奨形式や制約の表現方法が異なる。特定のバージョンに対応するだけでも相当な労力が必要になる。そのため、node を指定すれば、そのエンティティが表す JSON をシンプルに出力するほうが、誰にとっても理解しやすいと判断した。具体的な値ではなく、データ型を文字列として属性名に付与することで、JSON として整合性のある形で出力できる。以上の理由から、現状の json-modeler では JSON Schema 出力には対応していない。

8.3 JSON データが大きくなりすぎたらどうするか

この問題は確かに存在する。具体的に、MongoDB でも 1 ドキュメント（1 つの JSON 相当）のサイズには上限がある。しかし、モデリングの初期段階から JSON のサイズを気にするのは良い方法ではない。特に、モデリングの初期では、エンティティとリレーションシップの抽出に専念すべきである。Graph Script で

```
[A] -» [B] : include
```

と記述したとしても、後になって [A] の JSON が大きくなりすぎたり、[B] のライフサイクルや生成タイミングが異なることが判明する場合がある。その場合は、

```
[A] <- [B] : ref
```

に切り替え、別の JSON として管理すればよい。include/ref の判断は、意味 → 実装の順で行うべきであり、最初から実装上の制約に引きずられる必要はない。実際、Graph Script を読み込み後、include から ref への切り替えは、json-modeler でもサポートしている。ref により分離された JSON は、MongoDB のような、JSON 系 DB では、別 collection で管理する。RDB で言うと、別テーブルを作成して管理するということになる。

9 Enjoy Data Modeling

データモデリングは進化する。これまでの認識では対応できない事象に出会うことで、改善点が見つかり、データモデルやモデリング手法を改善していくことができる。だからこそ、問題を探すために、どんどんモデリングしてみるべきだ。モデリングするたびに新たな発見がある。データモデリングは楽しい。もっとモデリングを楽しもう。