

そのJavaScript、V8が泣いています。
V8の気持ちを理解して書くパフォーマンス最適化

自己紹介

- 名前：西 悠太
- 所属：株式会社ダイニー
- TypeScriptが大好きです



V8とは？

V8は、Googleが開発するオープンソースのJavaScriptおよびWebAssemblyエンジンです。現代のウェブとサーバーサイドアプリケーションの根幹を支える技術と言えます。

主な使用例:

- Google Chrome: 実行エンジンとしてV8を採用
- Node.js: V8をベースにしたJavaScript実行環境
- Electron: VS Code, Slack, Discordなど、数多くのデスクトップアプリケーションが内部でV8を利用

私たちが書くJavaScriptコードのパフォーマンスは、このV8の働きに直接的に依存するため、その内部構造を理解することは、より高速なアプリケーションを開発する上で極めて重要です。

V8コンパイラの全体像

V8は「起動速度」と「実行速度」という相反する要求を両立させるため、複数のコンパイラを使い分ける段階的最適化戦略を採用しています。

全てのコードを一度に最高レベルで最適化するのではなく、コードの実行頻度、いわゆる「温度」に応じて、より高度なコンパイラへと処理を引き継いでいきます。

層	コンパイラ	役割
Tier 0	Ignition (インタプリタ)	即時実行 & プロファイル収集
Tier 1	Sparkplug (ベースライン)	高速な非最適化コンパイル
Tier 2	Maglev (中間最適化)	バランスの取れた高速最適化
Tier 3	TurboFan (最適化)	最高レベルのパフォーマンス

すべての最適化の基礎: Ignition

Ignitionの役割

Ignitionは、V8のインタプリタです。JavaScriptコードを受け取り、可能な限り速く実行を開始します。

これを実現するために、JavaScriptのソースコードをマシンが直接実行するマシン語ではなく、**バイトコード**と呼ばれる中間表現に変換します。

このバイトコードは、Ignitionが内蔵する**レジスタベースの仮想マシン**によって解釈・実行されます。

Ignitionの最重要任務

Ignitionは単にコードを実行するだけではありません。

後の最適化コンパイラが最高のパフォーマンスを引き出すための**偵察**として、実行時のコードの振る舞いを詳細に記録します。

このプロファイル情報は **Feedback Vector** と呼ばれ、以下のような情報を含みます。

- **型情報**: 関数に渡された引数や、変数に代入された値がどのような型（数値、文字列、オブジェクトなど）であったか
- **オブジェクト形状**: オブジェクトがどのようなプロパティを、どのような順序で持っていたか（Hidden Class）
- **実行回数**: 関数が何回呼び出されたか、ループが何回繰り返されたか

Ignitionのキーテクノロジー①：Hidden Class

Hidden Class (V8内部名: Map) は、V8がオブジェクトの内部構造 (shape) を管理する仕組みです。
(一般的な文献では「shape」と表現されることもあります)

JavaScriptのオブジェクトは動的にプロパティを追加できますが、V8はプロパティの「名前と追加順序」(および属性/アクセサの種別) が同じオブジェクトを「同じ形状」とみなし、内部的に同じ Map を割り当てます。

この仕組みにより、V8はプロパティの格納場所 (in-object かプロパティ配列か、そのオフセット) を素早く特定できます。

オブジェクト生成後にプロパティを追加しても、全インスタンスが“同じ順序・同じ有無”で追加される限りは、同じ Map の遷移チェーンを辿るため最適化は維持されます。問題になるのは、インスタンスごとに追加順序や有無が揺れる場合や、`delete` などで形状を壊して辞書モードへ移行する場合で、こうした不安定さが IC を多相化させ、最適化効率を下げます。

Ignitionのキータクノロジー②: Elements Kind

Hidden Classがオブジェクトの形状を扱うのに対し、Elements Kindは配列の要素を効率的に扱うための仕組みです。V8は配列の中身に応じて、内部的な表現方法（Elements Kind）を動的に最適化します。

主な種類と速度順:

1. `PACKED_SMI_ELEMENTS`: 隙間なく、小さな整数(SMI)のみが格納されている状態
2. `PACKED_DOUBLE_ELEMENTS`: 浮動小数点数が含まれる状態
3. `PACKED_ELEMENTS`: 任意のJS値が格納される状態（数値は `HeapNumber` として格納）

`PACKED` (密) vs `HOLEY` (疎):

- `[1, 2, 3]` のような隙間のない配列は `PACKED` となり高速です
- `[1, , 3]` のように要素が抜けている配列は `HOLEY` となり低速になります 補足: `arr[1] = undefined` は“穴”ではありません。 `delete arr[1]`、離れたインデックスへの代入（例: `arr[9] = 1`）、`length` の先行拡張などが“穴”を作ります。

V8は可能な限り特化度の高い Elements Kind を維持しようとしませんが、型の異なる要素が追加されると、より一般的な種類へと基本的に一方向の遷移（降格）が発生します（同じ配列は原則戻りません。新しい配列を生成すれば `PACKED` にできます）。

Ignitionのキーテクノロジー③：Inline Cache (IC)

Inline Cache (IC) は、Hidden Classを利用してプロパティアクセスを劇的に高速化するキャッシュ機構です。

仕組み: 初回アクセス時に、オブジェクトのHidden Classとプロパティのメモリ上の位置（オフセット）をセットでキャッシュします。2回目以降は、同じHidden Classであれば検索をスキップし、直接オフセットを参照します。

状態:

- Monomorphic (単一形状): 最も理想的で最速
- Polymorphic (多形状): 2～4種類の形状を扱う。少し遅くなる
- Megamorphic (超多形状): 5種類以上の形状を扱う。ICによる最適化を諦め、大幅に低速化する

常識を覆した超高速コンパイラ: Sparkplug

Sparkplugの設計思想

Sparkplugは、V8 v9.1で導入された**非最適化**（ベースライン）コンパイラです。その核心的な思想は、従来のコンパイラの常識を完全に無視することにあります。

通常のコンパイラはソースコードをAST（抽象構文木）やIR（中間表現）に変換し、それを最適化してからマシン語を生成します。しかしSparkplugは、**Ignitionが生成したバイトコードから直接マシンコードを生成**することで、これらのプロセスを全て省略し、驚異的なコンパイル速度を実現します。 補足: SparkplugはIRを生成せず、最適化は基本的に行いません（ごく局所的なピープホール最適化程度）。

Sparkplugの内部実装と利点

Sparkplugの実装は、コンパイラ全体が実質的に巨大なswitch文を含む単一のループとして構成されています。各バイトコード命令に対し、事前に用意された固定のマシンコード生成関数を呼び出すだけ、という極めてシンプルな構造です。

このアプローチが効果的なのは、**多くのJavaScriptコードは数回しか実行されない**という現実があるためです。複雑な最適化にかかる時間が、それによって得られる実行時間の短縮を上回ることが多いため、Sparkplugは「変換による高速化」以外の最適化を大胆に切り捨てています。技術的には、ループ検出のための軽い前処理パスを経てからコード生成する実装（2パス）だった時期もありますが、本質はバイトコードの線形走査による直接コード生成です。実行速度向上の主因は、インタプリタ特有のオペランドのデコードや次バイトコードへのディスパッチといったオーバーヘッドを、事前に機械語へ前取りすることで取り除ける点にあります。また、Sparkplugはインタプリタ互換のスタックフレームを維持し、多くの処理を `builtins` に委譲するため、デバッガ/プロファイラ/例外スタック/OSR（最適化・脱最適化）の連携が容易です。

バランス型の現実的な最適化: Maglev

Maglevの設計思想：「十分に良いコードを、十分に速く」

Maglevは、SparkplugとTurboFanの間を埋める、高速な中間最適化コンパイラです。

Maglevは、Ignitionが収集したFeedback Vectorを初めて本格的に活用し、低コストで効果の高い最適化を実施します。

Maglevの主要技術①：SSAと制御フローグラフ

SSA (静的単一代入) 形式: MaglevはIRとしてSSA形式を採用しています。この形式では各変数が一度だけ代入されるため、データフローの解析が劇的に簡単になり、定数伝播や不要コード削除といった基本的な最適化が、複雑な解析なしに実現できます。

制御フローグラフ (CFG): コードの実行経路（ループや条件分岐）をグラフとして構築することで、基本的な最適化（例: 定数伝播や不要コード削除）を低コストで適用しやすくします。コンパイル時間を抑えるため、高度なループ変換は目的外です。

Maglevの主要技術②：型フィードバックと表現選択

型フィードバックによる最適化: Ignitionが収集した型情報を活用し、オブジェクトの形状(Hidden Class/Map)が安定しているプロパティアクセスに対しては、形状チェック（例: CheckMap）とオフセット読み取り（例: LoadField）に特化した効率的なコードを生成します。

表現選択 (Representation Selection) : 数値などの値について最適な内部表現（例: Smi/整数、浮動小数点、タグ付き/非タグ値）を選び、必要に応じてアンボックス/リボックスを行うことで、演算やレジスタ渡しを高速化します。

最高峰の最適化コンパイラ: TurboFan

TurboFanの役割

TurboFanは、V8で最も高度な最適化を担当するコンパイラです。主に実行頻度が高い「ホット」な関数に適用されます。近年は **Profile-Guided Tiering** により、特性に応じて早期にTurboFanへ直接昇格させるケースもあります。

TurboFanのコア技術①：中間表現(IR)アーキテクチャ

TurboFanは、最高レベルの最適化を実現するために、歴史的な **Sea-of-Nodes** と、近年本流になりつつある **Turboshift** の2系統のIRを背景に持ちます。

Sea-of-Nodes は、データフローと制御フローを単一のグラフで表現するIRです。この構造は、命令の並べ替えに最大限の柔軟性をもたらし、強力な大域的最適化を可能にしますが、その複雑さからコンパイル時間が長くなるという性質がありました。

それに対し **Turboshift** は、より伝統的な制御フローグラフ(CFG)をベースにしたIRです。最初にコードの実行経路を固定し、そのブロック内で最適化を行うため、コンパイルのオーバーヘッドが大幅に削減されます。V8は、JavaScriptの動的な性質上、Sea-of-Nodesの理論的な利点が常に活かせるとは限らないという経験則から、より実用的で高速なTurboshiftへの移行を進めています。現在、Wasmは大部分がTurboshift化済みで、JavaScript側もLoweringや最適化の多くが段階的にTurboshiftへ移行しています。

TurboFanのコア技術②：投機的最適化と脱最適化

投機的最適化は、TurboFanの真骨頂です。Feedback Vectorに基づき、「この変数は常に数値である」「この条件分岐は常にtrueである」といった **予測（賭け）** を行います。

この予測が的中している限り、型チェックなどを省略した非常に高速なコードが実行されます。もし予測が外れた場合、**脱最適化 (Deoptimization)** という安全装置が作動し、実行中のコードを即座に安全な下位のコンパイラのコードに戻します。これにより、安全性を担保しつつ、動的言語であるJavaScriptで静的言語並みの最適化を可能にしています。

TurboFanの高度な最適化技術

TurboFanは、前述のコア技術を基盤に、多数の高度な最適化技術を駆使します。

グローバル値番号付け (GVN): プログラム全体で冗長な計算を検出し、削除します。

数値範囲解析: 数値の取り得る範囲を解析し、分岐や境界チェックを削減します。

強度低減: 除算を乗算に、乗算をシフト演算に置き換えるなど、高コストな演算を低コストなものに変換します。

エスケープ解析: オブジェクトの逃避性に基づき、割り当てのスタック化や割り当て自体の削除を行います。

高度なループ最適化: ループ不変式の移動、ループ融合など。JS一般の自動ベクトル化は限定的で、主にWasm/一部の組み込みで有効です。

TurboFanの脱最適化の限界

TurboFanには重要な安全装置があります：同じ関数で脱最適化が多発すると、実装依存のヒューリスティックにより再最適化が抑制・遅延される場合があります（閾値や挙動はV8のバージョンやフラグで変動します）。

これは最適化と脱最適化の無限ループを防ぐための機構で、型が頻繁に変わるコードでは最適化されない状態が続きます。

2024年の新技術: Profile-Guided Tiering

IntelとGoogle V8チームが共同開発したProfile-Guided Tieringは、関数ごとに最適なコンパイラ戦略を選択します。

- 頻繁に使用される関数は早期にTurboFanへ直接昇格
- 脱最適化が多い関数は遅延ティアリング戦略を適用
- Speedometer 3で約5%の性能向上を実現 (Intel Core Ultra Series 2)

V8に好かれるコードの書き方

1: オブジェクトの形状を保持する

なぜ重要か？: オブジェクトの形状(Hidden Class/Map)が安定していると、**Inline Cache(IC)** が有効に機能し、プロパティアクセスが高速化されます。生成後にプロパティを追加しても、全インスタンスで“同じ順序・同じ有無”なら同じMap遷移を辿れるため最適化は維持されます。問題は、インスタンスごとに追加順序や有無が揺れる、`delete` が混じる等で形状が不安定になる場合で、ICが多相化して性能が低下し、場合によっては辞書モードへ移行します。

✖ Before: 形状が変化するコード

このコードでは、`user` は生成後に段階的にプロパティを追加しています。遷移自体は正常ですが、実運用でインスタンスごとに追加順序や有無が揺れるとICが多相化しやすく、最適化効率が落ちます。

```
const user = { id: 1 };  
user.name = "Taro";  
user.isAdmin = false;
```

✓ After: 形状が安定したコード

こちらのコードでは、`User` インスタンスは常に同じ形状を持つため、V8は安心して最適化できます。

```
class User {  
  constructor(id, name) {  
    this.id = id;  
    this.name = name;  
    this.isAdmin = false;  
  }  
}  
  
const user1 = new User(1, "Taro");  
const user2 = new User(2, "Jiro");
```

2: 型の一貫性を維持する

なぜ重要か？: 変数や引数の型が安定していると、TurboFanの**投機的最適化**が成功しやすくなります。V8は「この関数はきっと次も同じ型で呼ばれるだろう」と予測して最適化するため、その予測を裏切らないことが重要です。

✖ Before: 型が不安定なコード

このコードでは、`add` 関数に数値と文字列という異なる型が渡されています。これにより、V8のInline CacheはPolymorphicまたはMegamorphicな状態に陥り、型を毎回チェックする必要性が生まれるため、最適化の効率が著しく低下します。

```
function add(a, b) {  
  return a + b;  
}  
  
add(1, 2);  
add("a", "b");
```

✓ After: 型が一貫したコード

こちらのコードでは、関数を数値用と文字列用に分離しています。これにより、各関数は常に同じ型 (Monomorphic) の引数を受け取るため、V8は安心して型に特化した最適化を行うことができます。

```
function addInt(a, b) {  
  return a + b;  
}  
  
function concatStr(a, b) {  
  return a + b;  
}  
  
addInt(1, 2);  
concatStr("a", "b");
```

3: 例外による脱最適化を避ける

なぜ重要か?: 歴史的に、古いV8では `try...catch` があるだけで最適化が阻害されましたが、現代のV8 (TurboFan) ではこの挙動は改善されています。 `try...catch` ブロックの存在自体は、もはや最適化の妨げにはなりません。

しかし、重要なのは、実際に例外がスローされ `catch` されると、それが「予測不能な事態」と見なされ、高確率で脱最適化（低速なコードへのフォールバック）を引き起こすという点です。

ループ内など「ホット」な箇所で脱最適化が頻発すると、V8はその関数の最適化を最終的に諦めてしまう可能性があります。

✖ Before: 例外に頼った処理

ループ内でプロパティが存在しない可能性を `try...catch` で処理すると、`catch` に入るたびに脱最適化が起きるリスクがあります。

```
function getNames(users) {  
  const names = [];  
  for (const user of users) {  
    try {  
      // user.profile が無い場合に TypeError が発生  
      names.push(user.profile.name);  
    } catch (e) {  
      names.push('Unnamed');  
    }  
  }  
  return names;  
}
```

✓ After: 事前チェックによる安定した処理

`try...catch` を使わず、プロパティの存在を事前にチェックすることで、例外の発生と脱最適化のコストを完全に回避します。

```
function getNames(users) {  
  const names = [];  
  for (const user of users) {  
    const name = user.profile?.name ?? 'Unnamed';  
    names.push(name);  
  }  
  return names;  
}
```

4: インライン化されやすい関数を心がける

なぜ重要か？: V8は、小さな関数を呼び出し元に埋め込む**インライン化**で、関数呼び出しのコストを削減します。しかし、その判断は「何行以内」といった単純なものではありません。V8は関数のサイズ、呼び出し頻度、型の安定性などを総合的に評価し、インライン化の可否を判断します。V8にとって判断が難しい、大きすぎる・複雑すぎる関数はインライン化が見送られ、パフォーマンス向上の機会を逃してしまいます。

✖ Before: 大きく、複数の責任を持つ関数

データ加工、検証、整形といった複数の役割を担っており、V8にとってインライン化の判断が難しい状態です。

```
function processUser(user) {  
  const fullName = `${user.lastName} ${user.firstName}`;  
  if (fullName.length > 20) {  
    console.error('Name is too long');  
    return null;  
  }  
  return { id: user.id, fullName };  
}
```

✓ After: 小さく、単一責任の関数に分割

役割ごとに小さな関数に分割することで、各関数はV8にインライン化されやすくなります。特に `getFullName` のような小さく純粋な関数は、インライン化の最有力候補です。

```
function getFullName(user) {  
  return `${user.lastName} ${user.firstName}`;  
}
```

```
function validateName(name) {  
  return name.length <= 20;  
}
```

```
function processUser(user) {  
  const fullName = getFullName(user);  
  if (!validateName(fullName)) {  
    console.error('Name is too long');  
    return null;  
  }  
  return { id: user.id, fullName };  
}
```

5: 配列の型を固定する (Elements Kind)

V8の気持ち（なぜ重要か？） : V8は、配列の要素の型に応じて内部的に最も効率的な表現（ `Elements Kind` ）を選択します。要素がすべて整数であれば非常に高速な整数の配列として扱いますが、途中で1つでも浮動小数点数やオブジェクトが混ざると、より汎用的で低速な表現に変換（遷移）せざるを得ません。この遷移はV8にとって大きな負担です。

✖ Before: 配列の型が途中で変わる

最初は整数だけだった配列に、後から浮動小数点数を追加しています。この瞬間に `Elements Kind` のコストが高い遷移が発生します。

```
const numbers = [1, 2, 3]; // PACKED_SMI_ELEMENTS (最速)

// この代入により、配列は PACKED_DOUBLE_ELEMENTS (低速) に変換される
numbers.push(4.5);

// さらにオブジェクトを追加すると PACKED_ELEMENTS (さらに低速) になる
numbers.push({});
```

✓ After: 配列の型が一貫している

配列の型を一貫させることで、`Elements Kind` の遷移を防ぎ、V8は最速の状態での処理を続けることができます。

```
// 整数配列
const integers = [1, 2, 3];

// 浮動小数点数配列
const doubles = [1.1, 2.2, 3.3];

// オブジェクト配列
const objects = [{}, {}];
```

注意: `-0`、`NaN`、`Infinity` を 1 つでも含めると `..._DOUBLE_ELEMENTS` に遷移します。整数配列を維持したい場合は初期化時に値を正規化/バリデーションしましょう。

「V8の気持ち」と私たちのコード

これまでのTIPSを、V8の内部動作と結びつけて整理してみましょう。

V8の気持ち（内部動作）

私たちが書くべきコード

Inline Cache を効かせたい

オブジェクトの形状を一定に保つ

投機的最適化を成功させたい

変数や引数の型を安定させる

脱最適化を避けたい

例外の発生を防ぐ事前チェックを行う

インライン化しやすくしたい

小さく単一責任の関数に分割する

Elements Kindの遷移を防ぎたい

配列に入れる要素の型を固定する

まとめ

V8は Ignition, Sparkplug, Maglev, TurboFan という多層コンパイラ構造で、起動速度と実行速度を両立させています。

開発者として重要なのは、V8の気持ち、すなわち「予測しやすいコードを好む」という性質を理解することです。

オブジェクトの形状や配列の型を安定させ、関数の引数の型を揃え、例外に頼らない安定した処理を心がけ、関数を小さく保つこと。これらの積み重ねが、V8があなたのコードを最大限に高速化するための鍵となります。

ご清聴ありがとうございました

本日のスライドは下記のリポジトリで公開しています。

内容の修正・改善など、お気軽にPull Requestをお送りください。

11/30の関西のフロントエンドカンファレンスでも登壇するので、そこでもお会いしましょう！

https://github.com/riya-amemiya/amemiya_riya_slide_data/tree/main/frontend_conf_tokyo_2025

- XやGitHubなど: <https://riya-amemiya-links.tokidux.com/>



このスライドは CC BY-SA 4.0 でライセンスされています。

より自由な翻訳を可能にするため、翻訳は例外的に CC BY 4.0 での配布が許可されています。

Required Attribution: Riya Amemiya (<https://github.com/riya-amemiya>)