# 設計原則

このドキュメントを作成したのは、React が何をして何をしないのか、そして開発理念がどのようなものなのかをより理解できるようにするためです。私たちはコミュニティへの貢献を見ることにわくわくしています。しかしこれらの原則の 1 つ以上に違反する道を選ぶことはありません。

**注意:**
このドキュメントは React を強く理解していることを前提としています。React のコンポーネントやアプリケーションではなく、**React 自体**の設計原則について説明しています。
React の紹介については、代わりに React の流儀を調べてください。

**コンポジション**
React の主な機能はコンポーネントのコンポジションです。異なる人々によって書かれたコンポーネントは一緒にうまく動くべきです。コードベース全体に波及する変更を引き起こすことなく、コンポーネントに機能を追加できるということが重要です。
たとえば、コンポーネントを使用する側を変更せずに、コンポーネントにローカル state を導入することが可能であるべきです。同様に、必要に応じて初期化と終了処理を任意のコンポーネントに追加できるべきです。
コンポーネントで state またはライフサイクルメソッドを使用することについて「悪い」ことは何もありません。他の強力な機能と同様に適度に使用する必要がありますが、私たちはそれらを削除するつもりはありません。それどころか、それらは React を有用にするものとして不可欠な部分であると思います。将来的にはより関数型的なパターンを使用可能にするかもしれませんが、ローカルステートとライフサイクルメソッドの両方がそのモデルの一部になるでしょう。
コンポーネントはしばしば「単なる関数」と表現されますが、私たちの見解では、それらは有用であるために単なる関数以上である必要があります。React では、コンポーネントは組み合わせ可能な動作を記述します。そしてこれには、レンダリング、ライフサイクル、および state が含まれます。Relay のようないくつかの外部ライブラリは、データの依存関係を記述するといったような他の責任をコンポーネントに付け加えます。それらのアイデアが何らかの形で React に改めて取り込まれるということも有り得ます。

**共通の抽象化**
一般的に私たちはユーザー側で実装できる機能の追加をできるだけ避けます。使用されない無駄なライブラリコードであなたのアプリを肥大化させたくありません。ただし、これには例外があります。
たとえば、もし React がローカル state やライフサイクルメソッドをサポートしなかったなら、人々はそれらに対して独自の抽象化を作成したことでしょう。複数の抽象化が競合している場合、React はどちらかの利用を勧めたりどちらか一方のみにある特性を利用したりすることができなくなります。最小公倍数の機能でやっていかなければならないのです。
これが、React 自体に機能を追加することがある理由です。多くのコンポーネントが互換性のない、または非効率的な方法で特定の機能を実装していることに気づいたら、React にそれを織り込むかもしれません。しかし私たちは気軽にはそれをしません。それをするときは抽象化レベルを上げることがエコシステム全体に利益をもたらすと確信しているからです。state、ライフサイクルメソッド、クロスブラウザのイベント正規化などがその好例です。
私たちは常にそのような改善提案をコミュニティーと話し合います。それらの議論のいくつかは React イシュートラッカーの “big picture” ラベルで見つけることができます。

**避難ハッチ**
React は実用主義的です。それは Facebook で書かれた製品のニーズによって推進されています。関数型プログラミングなど、まだ完全には主流になっていない、いくつかのパラダイムによって影響を受けますが、さまざまなスキルや経験レベルを持つ幅広い開発者がアクセスしやすいことがプロジェクトの明確な目標です。
私たちが好まないパターンを非推奨にしたい場合は非推奨にする前に、既存のすべてのユースケースを検討し、コミュニティに代替案について教育することが私たちの責任です。アプリを構築するのに有用なパターンを宣言的に表現するのが難しい場合は、命令的な API を提供します。多くのアプリで必要と思われるものに最適な API を見つけられないときは、一時的な標準未満の作業用 API を提供します（後で取り除くことが可能で、将来の改善の余地がある場合に限ります）。

**安定性**
私たちは API の安定性を重視しています。Facebook には、React を使った 5 万以上のコンポーネントがあります。Twitter や Airbnb を含む他の多くの会社もまた React のヘビーユーザーです。これが私たちが公開された API や振る舞いを変更することに大抵消極的である理由です。
しかし、「何も変わらない」という意味での安定性を人々は過大評価している、と私たちは考えます。それはすぐに停滞に変わってしまいます。代わりに、「本番環境では頻繁に使用され、何かが変わったときには明確な（できれば自動化された）移行方法がある」という意味での安定性を優先します。
あるパターンを非推奨にするときは、Facebook 内部でその使用法を調べ、非推奨警告を追加します。これにより変化の影響が評価できます。変更が時期尚早であり、準備が整うところまでコードベースを持っていくのにさらに戦略的に検討する必要があると分かった場合、変更を差し戻すことがあります。
変更があまり破壊的でなく、移行戦略がすべてのユースケースで実行可能であると確信している場合は、非推奨の警告をオープンソースコミュニティに公開します。私たちは Facebook 以外の多くの React ユーザーと密接に連絡を取り合っており、人気のあるオープンソースプロジェクトを監視し、非推奨警告の修正を手引きしています。
Facebook の React コードベースのサイズが非常に大きいことを考えると、社内での移行が成功することは、他の企業でも問題ないことを示す良い指標となります。それでも、時々人々は私達が考えていなかった新たなユースケースを指摘するため、私たちはそれらのための避難ハッチを追加したり、アプローチを再考したりします。
正当な理由がない限り、私たちは何も非推奨にすることはありません。非推奨警告は時々イライラさせられるということを認識していますが、それを追加するのは、私たちとコミュニティの多くの人々が価値あると考える改善と新機能追加のための道が、非推奨化によって開かれるからです。
たとえば、React 15.2.0 で未知の DOM props に関する警告を追加しました。多くのプロジェクトがこの影響を受けました。しかしこの警告を修正することは、React にカスタム属性のサポートを導入できるようにするために重要でした。私たちが追加するすべての非推奨化の背景には、このような理由があります。
非推奨警告を追加すると、現在のメジャーバージョンの残りの部分では警告のまま残り、次のメジャーバージョンでは動作が変更されます。繰り返し行われる手作業が多い場合は、変更の大部分を自動化する codemod スクリプトをリリースします。Codemod を使用すると、大規模なコードベースでも移行を滞りなく進めることができます。それらを使用することをお勧めします。
私たちがリリースした codemod は、react-codemod リポジトリで見つけることができます。

**相互運用性**

私たちは既存のシステムとの相互運用性と段階的な導入に高い価値を置いています。Facebook は巨大な非 React のコードベースを持っています。Facebook の Web サイトでは、XHP と呼ばれるサーバーサイドのコンポーネントシステム、React よりも前に開発された内部 UI ライブラリ、そして React 自体を組み合わせて使用しています。私たちにとって重要なことは、どの製品チームでも、賭けるようにコードを書き換えるのではなく、小さな機能に対して React を使い始めることができるということです。

これが、React がミュータブルなモデルを扱うための避難ハッチを提供し、他の UI ライブラリと一緒にうまく機能しようとする理由です。既存の命令的な UI を宣言型コンポーネントにラップすることも、その逆も可能です。これは段階的な採用には不可欠です。

**スケジューリング**

コンポーネントが関数として記述されていても、React を使うときはそれらを直接呼び出さないでください。すべてのコンポーネントは何をレンダリングする必要があるかの説明書きを返し、その説明書きには `<LikeButton>` のようなユーザー作成のコンポーネントと `<div>` のようなプラットフォーム固有のコンポーネントの両方を含めることができます。将来のある時点で `<LikeButton>` を「展開」して、コンポーネントのレンダリング結果に従って UI ツリーを実際に再帰的に変更するのは、React の責務です。

これは微妙な違いですが強力なものです。あなたがコンポーネント関数を呼び出さずに React が呼び出します。それは React は必要に応じて呼び出しを遅らせる権限があるということを意味します。現在の React の実装ではツリーを再帰的に調べて、1 回のイベントループの間に更新されたツリー全体のレンダリング関数を呼び出します。しかし、将来的にはフレームのドロップを避けるためにいくつかの更新を遅らせるかもしれません。

これは React の設計の共通テーマです。いくつかの人気のあるライブラリは、新しいデータが利用可能になったときに計算が実行される「プッシュ」アプローチを実装しています。しかし React は、計算が必要になるまで遅らせることができる「プル」アプローチを採用しています。

React は汎用的なデータ処理ライブラリではありません。ユーザーインターフェイスを構築するためのライブラリです。アプリ内において React は、どの計算が今すぐ必要でどの計算がそうでないのかを知ることができる特殊な位置づけにある、と私たちは考えています。

何かが画面外にある場合は、それに関連するロジックを遅らせることができます。データがフレームレートよりも早く到着する場合は、合体してバッチ更新することができます。フレームを落とさないように、重要度の低いバックグラウンド作業（ネットワークからロードされたばかりの新しいコンテンツのレンダーなど）よりも、ユーザーの操作による操作（ボタンクリックによるアニメーションなど）を優先できます。

念のために言うと、今の時点ではこれらの可能性は実現していません。しかし、このようなことを自由に行いたいということが、私たちがスケジューリングを制御したい理由であり、`setState()` が非同期である理由です。概念的には、`setState()` を「更新のスケジュール」だと考えています。

ユーザが関数型リアクティブプログラミングのいくつかのバリエーションで一般的な「プッシュ」ベースのパラダイムでビューを直接構成させた場合、スケジューリングに対する制御を得るのが難しくなります。「糊付け部分」のコードは私たちが管理したいのです。

React の主な目標は、React 内部に戻る前に実行されるユーザーコードの量を最小限にすることです。これは React が UI について知っていることに従ってスケジュールしたり小分けに作業を分割したりする能力を保持することを保証します。

React は完全に「リアクティブ」であることを望んでいないため、React は “Schedule” と呼ばれるべきだったというチーム内の冗談があります。

**開発体験**

良い開発者経験を提供することは私達にとって重要です。

たとえば、Chrome と Firefox で React コンポーネントツリーを調べることができる React DevTools をメンテしています。私たちは、それが Facebook エンジニアとコミュニティの両方に大きな生産性向上をもたらしていると聞いています。

私たちは開発者向けの有用な警告を提供するために、一層の努力をするようにしています。たとえば、React は開発中にブラウザが理解できない方法でタグをネストした場合、または API で一般的な入力ミスをした場合に警告します。開発者向けの警告とそれに関連するチェックが、React の開発版が製品版より遅い主な理由です。

Facebook で内部的に見られる使用パターンは、よくある間違いとは何か、そしてそれらを早期に防ぐ方法を理解するのに役立ちます。私たちが新しい機能を追加するとき、私たちはよくある間違いを予想してそれらについて警告しようとします。

私たちは開発者の体験を向上させる方法を常に探しています。体験をさらに良くするために、ぜひ皆さんの提案を聞き、貢献を受け入れたいと思っています。

**デバッグ**

問題が発生した場合は、コードベースでその間違いの原因を突き止めるための “パンくず” (breadcrumb) を作成することが重要です。React では、props と state がそのようなパンくずです。

画面に問題がある場合は、React DevTools を開いてレンダーを担当するコンポーネントを見つけ、次に props と state が正しいかどうかを確認できます。そうであれば、問題はコンポーネントの `render()` 関数、または `render()` によって呼び出される関数にあることがわかります。これで問題が切り分けられました。

state が間違っている場合、問題はこのファイル内の `setState()` の呼び出しの 1 つによって引き起こされていることがわかります。これも、通常は 1 つのファイル内に `setState()` の呼び出しは数回しかないため、検索と修正が比較的簡単です。

props が間違っている場合は、インスペクタでツリーを上にたどり、悪い props を渡して最初に「井戸に毒を入れた」犯人のコンポーネントを探します。

任意の UI を生成した元データまで props と state の現在値を使って追跡できるというこの能力は React にとって非常に重要です。state がクロージャやコンビネータに「閉じ込め」られておらず、React で直接利用できることは明示的な設計目標です。

UI は動的ですが、props と state の同期的な `render()` 関数により、デバッグ作業が単なる当て推量から、退屈ながら有限の手順になると信じています。これにより複雑なアニメーションのようないくつかのユースケースがより困難になりますが、React ではこの制約を保持したいと思います。

**設定**

グローバルな実行時設定オプションは問題があることがわかりました。

例えば、`React.configure(options)` や `React.register(component)` のような関数を実装することが時々要求されます。しかし、これは複数の問題を引き起こし、私たちはそれらに対する良い解決策を知りません。

誰かがサードパーティのコンポーネントライブラリからそのような関数を呼び出すとどうなりますか？ ある React アプリに別の React アプリが埋め込まれていて、それらの望ましい設定に互換性がない場合はどうなりますか？ サードパーティコンポーネントは特定の設定をどのように必須にしますか？ グローバル設定はコンポジションではうまく機能しないと考えています。コンポジションは React の中心であるため、コードでグローバル設定を提供しません。

ただし、ビルドレベルでグローバル設定をいくつか提供します。たとえば、開発ビルドと本番ビルドを別々に提供しています。将来プロファイリングビルドを追加するかもしれませんし、また、他のビルドフラグの検討を受け入れています。

**DOM を超えて**

私たちは React の価値を、バグの少ないコンポーネントを書いてうまく構成することができるという点に見ています。DOM は React のオリジナルのレンダーターゲットですが、React Native は Facebook とコミュニティの両方において同じくらい重要です。

レンダラーに依存しないことは React の重要な設計上の制約です。それは内部表現にいくらかのオーバーヘッドを追加します。その一方で、コアへの改善はすべてのプラットフォームに行きわたります。単一のプログラミングモデルを持つことで、プラットフォームではなく製品を中心にエンジニアリングチームを形成できます。これまでのところ、そのトレードオフは私たちにとって価値があります。

**実装**

可能な限り洗練された API を提供しようとしています。しかし実装がエレガントであることにはそれほど関心がありません。現実の世界は完璧には程遠いので、ユーザーが醜いコードを書かなくて済むのであれば、合理的な範囲で醜いコードをライブラリに入れることを選びます。新しいコードを評価するときには、正しく、パフォーマンスが高く、優れた開発者体験を提供する実装を求めます。優雅さは二の次です。

私たちは賢いコードより退屈なコードを好みます。コードは使い捨てであり、しばしば変更されます。したがって、それが絶対に必要でない限り、新しい内部抽象化を導入しないことが重要です。移動、変更、削除が容易な冗長コードは、時期尚早に抽象化され変更が難しいエレガントなコードよりも優先されます。

**ツールへの最適化**

いくつかの一般的に使用される API は冗長な名前を持っています。例えば、`didMount()` や `onMount()` の代わりに `componentDidMount()` を使います。これは意図的です。目的は、ライブラリとのやり取りのポイントをよく見えるようにすることです。

Facebook のような大規模なコードベースでは、特定の API の使用を検索できることが非常に重要です。他と区別しやすい冗長な名前を大切にしています。特に、控えめに使用する必要がある機能についてはそれが重要です。例えば、`dangerouslySetInnerHTML` をコードレビューで見逃すことは難しいでしょう。

私たちは破壊的な変更を加える際に codemods に依存しているため、検索を最適化することも重要です。膨大な自動化された変更をコードベース全体に適用するのが簡単で安全であることを望みます。区別しやすい冗長な名前を使用すると、これを実現できます。同様に、他と区別可能な名前を使用すると、潜在的な誤検知を心配することなく、React の使用に関するカスタムの lint ルールを簡単に作成できます。

JSX も同様の役割を果たします。React では必須ではありませんが、美観上および実用上の理由から、Facebook で広く使用されています。

私たちのコードベースでは、JSX はそれらが React 要素ツリーを扱っているというツールへの明白なヒントを提供します。これにより、定数要素の巻き上げ、安全な lint および codemod 内部コンポーネントの使用などのビルド時の最適化を追加したり、JSX ソースの場所を警告に含めることができます。

**ドッグフーディング**

私たちはコミュニティーによって提起された問題に取り組むために最善を尽くします。しかし、私たちは Facebook **も**内部で経験している問題を優先する可能性があります。直感に反するかもしれませんが、私たちはこれこそがコミュニティが React に賭けることができる主な理由であると思います。

大量に内部で使用していることで、React が明日消えないという自信が得られます。React は Facebook の問題を解決するために Facebook で作成されました。それは Facebook に確かなビジネス価値をもたらし、多くの製品で使用されています。ドッグフーディングによって、私たちのビジョンが鮮明なままであり、私たちが今後も焦点を絞った方向性を持てるようになります。

これは、コミュニティが提起した問題を無視しているという意味ではありません。たとえば、Facebook 内部で依存していないにも関わらず、React に Web Components および SVG のサポートを追加しました。私達は積極的に皆さんの問題点を聞き、私たちの能力の及ぶ限りでそれらに対処します。コミュニティは React を私たちにとって特別なものにしており、私たちもお返しとして喜んでコミュニティに貢献したいと考えています。

Facebook で多くのオープンソースプロジェクトをリリースした後、みんなを同時に幸せにすることを試みるが焦点が不十分なプロジェクトを生み出してもうまく成長しないということを学びました。代わりに、少人数の観客を選んで満足させることに集中すると最終的には良い効果があることがわかりました。これこそまさに私たちが React を使って行ったことであり、そして今のところ Facebook 製品チームが遭遇した問題を解決することは、オープンソースコミュニティにもうまく還元されています。

このアプローチの欠点は、React の初期体験の良し悪しなど、Facebook チームが対処する必要がないことに十分に焦点を当てることができないことです。私たちはこのことを強く認識しており、以前にオープンソースプロジェクトで行ったのと同じ過ちを犯さずに、コミュニティのすべての人に利益をもたらすようなやり方でどのような改善ができるかを考えています。

このページを編集する