CSSレイアウト：Flexbox, Grid, レスポンシブデザイン

これまでの資料で、HTMLで構造を作り、CSSで見た目を整える方法を学んできましたね。しかし、要素を思った通りの場所に配置する「レイアウト」は、Web制作の中でも特に重要な、そして少し難しい部分です。

昔は `float` や `position` を駆使してレイアウトを組むのが一般的でしたが、現代ではもっと強力で柔軟なレイアウトシステムが登場しています。それが **Flexbox** と **Grid** です！

この資料では、これらの現代的なCSSレイアウト手法と、スマートフォンやタブレットなど、様々な画面サイズに対応するための **レスポンシブデザイン** について学んでいきます。これらをマスターすれば、複雑なレイアウトも効率的に、そして美しく組むことができるようになります。

1. Flexbox：一次元レイアウトの達人

Flexbox (Flexible Box Layout Module) は、主に **一次元のレイアウト**（横一列、または縦一列）を簡単に、そして柔軟に実現するための仕組みです。アイテムの配置、順序、間隔、伸縮などを細かくコントロールできます。ナビゲーションメニュー、ボタンの横並び、カード型コンテンツの配置など、様々な場面で活躍します。

Flexboxを使うには、まず親要素（**フレックスコンテナ**）に `display: flex;` を指定します。すると、その直下の子要素（**フレックスアイテム**）が横一列に並び、特別なプロパティで制御できるようになります。

```
<div class="flex-container">
  <div class="flex-item">アイテム1</div>
  <div class="flex-item">アイテム2</div>
  <div class="flex-item">アイテム3</div>
</div>
```

```
.flex-container {
  display: flex; /* これでFlexboxが有効になる */
  border: 2px solid blue; /* コンテナの範囲を分かりやすく */
  padding: 10px;
}

.flex-item {
  background-color: lightblue;
  border: 1px solid gray;
  padding: 20px;
  margin: 5px; /* アイテム間の隙間 */
}
```

フレックスコンテナに指定するプロパティ

- **`flex-direction`**: アイテムを並べる方向を指定します。
  - `row`（デフォルト）: 左から右へ横方向に並べる。
  - `row-reverse`: 右から左へ横方向に並べる。
  - `column`: 上から下へ縦方向に並べる。
  - `column-reverse`: 下から上へ縦方向に並べる。
- **`justify-content`**: 主軸（`flex-direction`で決まる方向）方向のアイテムの配置方法を指定します。
  - `flex-start`（デフォルト）: 開始位置に寄せる。
  - `flex-end`: 終了位置に寄せる。
  - `center`: 中央に寄せる。
  - `space-between`: 最初のアイテムは開始位置、最後のアイテムは終了位置に配置し、残りのアイテムは均等な間隔で配置。
  - `space-around`: 各アイテムの周りに均等な間隔を配置（両端のアイテムは半分の間隔になる）。
  - `space-evenly`: すべてのアイテムの間、および両端に均等な間隔を配置。
- **`align-items`**: 交差軸（主軸と垂直な方向）方向のアイテムの配置方法を指定します。
  - `stretch`（デフォルト）: コンテナの高さ（または幅）いっぱいにアイテムを伸ばす（アイテムに高さ/幅が指定されていない場合）。
  - `flex-start`: 交差軸の開始位置に寄せる。
  - `flex-end`: 交差軸の終了位置に寄せる。
  - `center`: 交差軸の中央に寄せる。
  - `baseline`: アイテム内のテキストのベースライン（文字の下線）を揃える。
- **`flex-wrap`**: アイテムがコンテナに収まりきらない場合に、折り返すかどうかを指定します。
  - `nowrap`（デフォルト）: 折り返さず、はみ出す（または縮む）。
  - `wrap`: 下（または右）に折り返す。
  - `wrap-reverse`: 上（または左）に折り返す。
- **`align-content`**: `flex-wrap: wrap;`で複数行になった場合の、行間の配置方法を指定します（`justify-content`の交差軸版のようなもの）。`align-items`と似ていますが、こちらは行全体を対象とします。
  - `stretch`（デフォルト）, `flex-start`, `flex-end`, `center`, `space-between`, `space-around`, `space-evenly`
- **`gap`, `row-gap`, `column-gap`**: アイテム間の隙間を指定します。`margin`を使うより簡単で正確な場合が多いです。

```
.flex-container {
  display: flex;
  height: 200px; /* align-itemsの効果を分かりやすくするため */
  background-color: #eee;

  /* よく使う組み合わせ例 */
  flex-direction: row; /* 横並び（デフォルト）*/
  justify-content: space-around; /* 横方向: 均等配置（周りに余白）*/
  align-items: center; /* 縦方向: 中央揃え */
  flex-wrap: wrap; /* 折り返しあり */
  gap: 10px; /* アイテム間の隙間を10pxに */
}
```

フレックスアイテムに指定するプロパティ

- **`order`**: アイテムの表示順序を数値で指定します。デフォルトは `0` で、小さい数値ほど先に表示されます。HTMLの構造を変えずに見た目の順序を変えられます。
- **`flex-grow`**: コンテナ内の余剰スペースを、他のアイテムと比較してどれだけ分配されるかの比率を指定します。デフォルトは `0`（伸びない）。`1` を指定すると、他の `flex-grow` が `0` のアイテムが使わないスペースをすべて使って伸びます。複数のアイテムに `1` を指定すると、余剰スペースを均等に分け合います。`2` を指定すると、`1` のアイテムの2倍伸びます。
- **`flex-shrink`**: コンテナのスペースが足りない場合に、他のアイテムと比較してどれだけ縮むかの比率を指定します。デフォルトは `1`（縮む）。`0` を指定すると縮まなくなります。
- **`flex-basis`**: アイテムの初期サイズ（主軸方向の幅または高さ）を指定します。`width` や `height` の代わりになります。`auto`（デフォルト）や `content`、具体的な長さ（`100px`, `30%`）を指定できます。
- **`flex`**: `flex-grow`, `flex-shrink`, `flex-basis` をまとめて指定するショートハンドプロパティです。よく使う値として `flex: 0 1 auto;`（デフォルト）, `flex: 1;`（`1 1 0%`）, `flex: auto;`（`1 1 auto`）, `flex: none;`（`0 0 auto`）などがあります。
- **`align-self`**: 特定のアイテムだけ、コンテナの `align-items` の設定を上書きして配置方法を指定します。（`auto`, `stretch`, `flex-start`, `flex-end`, `center`, `baseline`）

```
.flex-item {
  padding: 20px;
  border: 1px solid gray;
}

.item1 {
  order: 2; /* 2番目に表示 */
  background-color: lightcoral;
}
.item2 {
  order: 1; /* 1番目に表示 */
  flex-grow: 1; /* 他のアイテムが使わないスペースをすべて使う */
  background-color: lightgreen;
}
.item3 {
  order: 3; /* 3番目に表示 */
  flex-shrink: 0; /* 縮まないようにする */
  flex-basis: 150px; /* 初期幅を150pxに */
  background-color: lightskyblue;
}
.item4 {
  order: 4;
  align-self: flex-end; /* このアイテムだけ下に配置 */
  background-color: lightgoldenrodyellow;
}
```

開発者の視点: Flexboxは一次元レイアウトには非常に強力ですが、二次元（縦横両方）の複雑なレイアウトには次に出てくる Grid Layout の方が適している場合が多いです。とはいえ、多くのUIコンポーネント（ヘッダー、フッター、カードリスト、フォーム要素の配置など）はFlexboxで十分に、かつ簡単に実装できます。まず

はFlexboxをしっかりマスターしましょう。`justify-content` と `align-items` の組み合わせは頻繁に使うので、動きをよく理解しておくと良いです。

2. Grid Layout：二次元レイアウトの革命

CSS Grid Layout は、Webページを **行（Row）と 列（Column）からなる格子状（グリッド）** に分割し、その格子の中に要素を配置していく、**二次元レイアウト** のためのシステムです。Flexboxが一次元（線）のレイアウトを得意とするのに対し、Gridは縦横両方の配置を同時に、かつ非常に柔軟に制御できます。ページ全体のレイアウトや、複雑なコンポーネントの配置に適しています。

Grid Layoutを使うには、まず親要素（**グリッドコンテナ**）に `display: grid;` を指定します。そして、どのような格子を作るかを定義します。

```
<div class="grid-container">
  <div class="grid-item item1">ヘッダー</div>
  <div class="grid-item item2">サイドバー</div>
  <div class="grid-item item3">メインコンテンツ</div>
  <div class="grid-item item4">フッター</div>
</div>
```

グリッドコンテナに指定するプロパティ

- **`grid-template-columns`**：列の幅や数を定義します。
  - 長さ（`100px`, `10em`）
  - パーセンテージ（`50%`）
  - `auto`：コンテンツに応じた幅
  - **`fr` 単位**：利用可能なスペースを分割する比率（`1fr 2fr` なら1:2の比率で分割）。**Grid Layoutで非常によく使われます。**
  - `repeat(回数, サイズ)`：同じ定義を繰り返す（`repeat(3, 1fr)` は `1fr 1fr 1fr` と同じ）。
  - `minmax(最小値, 最大値)`：幅の最小値と最大値を指定（`minmax(100px, 1fr)`）。
- **`grid-template-rows`**：行の高さを定義します。指定方法は `grid-template-columns` と同様です。
- **`gap`**, **`row-gap`**, **`column-gap`**：グリッド線（トラック）間の隙間（溝）を指定します。Flexboxの `gap` と同じように使えます。
- **`grid-template-areas`**：グリッドの各セルに名前を付け、その名前を使ってアイテムを配置するエリアを定義します。視覚的に分かりやすいレイアウト指定が可能です。

```
.grid-container {
  display: grid;
  grid-template-columns: 1fr 3fr; /* 1列目は1fr, 2列目は3fr */
  grid-template-rows: auto 1fr auto; /* 1行目:自動, 2行目:残り全部, 3行目:自動 */
  grid-template-areas:
    "header header"   /* 1行目: headerエリアが2列分 */
    "sidebar main"    /* 2行目: sidebarエリアとmainエリア */
```

```
    "footer footer";  /* 3行目: footerエリアが2列分 */
  gap: 10px;
  min-height: 100vh; /* 画面の高さいっぱいに */
}
```

- **justify-items**: グリッドセル内のアイテムを、行方向（インライン軸）にどのように配置するかを指定します（全アイテムに適用）。（`start`, `end`, `center`, `stretch`（デフォルト））
- **align-items**: グリッドセル内のアイテムを、列方向（ブロック軸）にどのように配置するかを指定します（全アイテムに適用）。（`start`, `end`, `center`, `stretch`（デフォルト））
- **place-items**: `align-items` と `justify-items` をまとめて指定します（`align-items / justify-items`）。
- **justify-content**: グリッドコンテナ内のグリッド全体の配置を行方向に指定します（グリッド全体のサイズがコンテナより小さい場合）。Flexboxの同名プロパティと似ています。
- **align-content**: グリッドコンテナ内のグリッド全体の配置を列方向に指定します（グリッド全体のサイズがコンテナより小さい場合）。Flexboxの同名プロパティと似ています。
- **place-content**: `align-content` と `justify-content` をまとめて指定します（`align-content / justify-content`）。

グリッドアイテムに指定するプロパティ

- **grid-column-start**, **grid-column-end**: アイテムがどの列のグリッド線から始まり、どの列のグリッド線で終わるかを指定します。
- **grid-row-start**, **grid-row-end**: アイテムがどの行のグリッド線から始まり、どの行のグリッド線で終わるかを指定します。
- **grid-column**: `grid-column-start / grid-column-end` のショートハンド。`span n` を使うと「n個分のセルを占める」という指定もできます（例: `1 / span 2` は1番目の線から2つ分のセル）。
- **grid-row**: `grid-row-start / grid-row-end` のショートハンド。
- **grid-area**: アイテムを配置するエリア名を指定します（`grid-template-areas` で定義した名前）。または、`grid-row-start / grid-column-start / grid-row-end / grid-column-end` のショートハンドとしても使えます。

```
/* grid-template-areas を使う場合 */
.item1 { grid-area: header; background-color: lightcoral; }
.item2 { grid-area: sidebar; background-color: lightgreen; }
.item3 { grid-area: main; background-color: lightskyblue; }
.item4 { grid-area: footer; background-color: lightgoldenrodyellow; }

/* グリッド線で指定する場合（上記と同じレイアウト）*/
/* .item1 { grid-column: 1 / 3; grid-row: 1; } */
/* .item2 { grid-column: 1; grid-row: 2; } */
/* .item3 { grid-column: 2; grid-row: 2; } */
/* .item4 { grid-column: 1 / 3; grid-row: 3; } */
```

- `justify-self`: 特定のアイテムだけ、コンテナの `justify-items` の設定を上書きして行方向の配置を指定します。(`start`, `end`, `center`, `stretch`)
- `align-self`: 特定のアイテムだけ、コンテナの `align-items` の設定を上書きして列方向の配置を指定します。(`start`, `end`, `center`, `stretch`)
- `place-self`: `align-self` と `justify-self` をまとめて指定します (`align-self / justify-self`)。

開発者の視点: Grid Layout は非常に高機能で、最初は少し難しく感じるかもしれません。特に `fr` 単位と `grid-template-areas` は Grid の強力さを実感できる機能なので、ぜひ使ってみてください。ブラウザの開発者ツール（要素を検証）には Grid レイアウトを視覚的に確認できる機能があるので、それを活用すると理解が深まります。Flexbox と Grid は競合するものではなく、それぞれ得意なことが違います。ページ全体の大きなレイアウトは Grid で組み、その中のコンポーネント（ナビゲーションやカードリストなど）の内部レイアウトは Flexbox で組む、といった使い分けが一般的です。

3. Position：要素を自在に配置する（補足）

Flexbox や Grid が登場する前から使われているレイアウト手法として `position` プロパティがあります。これらも特定の場面では依然として有効です。

- `static` **（デフォルト）**: 通常の文書の流れに従って配置されます。`top`, `right`, `bottom`, `left`, `z-index` は効きません。
- `relative`: 通常の配置位置を基準として、`top`, `right`, `bottom`, `left` で相対的に位置をずらすことができます。元の位置にはスペースが残ります。`absolute` の基準点を作るためによく使われます。
- `absolute`: 通常の文書の流れから切り離され、最も近い `position: static` 以外の祖先要素（通常は `relative` や `absolute` が指定された親要素）を基準として、`top`, `right`, `bottom`, `left` で絶対的な位置を指定します。基準となる祖先要素がない場合は、`<body>` 要素が基準になります。元の位置にはスペースが残りません。
- `fixed`: 通常の文書の流れから切り離され、ブラウザの表示領域（ビューポート）を基準として、`top`, `right`, `bottom`, `left` で位置を固定します。スクロールしても位置が変わりません。ヘッダーやフッターの固定、モーダルウィンドウなどに使われます。
- `sticky`: 通常は `static` のように振る舞いますが、スクロールして特定の閾値 (`top`, `right`, `bottom`, `left` で指定）に達すると、`fixed` のようにその位置に固定されます。スクロール追従ヘッダーなどに便利です。

`position: relative`, `absolute`, `fixed`, `sticky` を指定した要素では、`z-index` プロパティで要素の重なり順（奥行き）を制御できます。数値が大きいほど手前に表示されます。

```
.parent {
  position: relative; /* .child の基準点 */
  width: 300px;
  height: 200px;
  border: 1px solid red;
}

.child {
  position: absolute;
  top: 10px; /* 親要素の上辺から10px */
```

```
  right: 10px; /* 親要素の右辺から10px */
  background-color: yellow;
  padding: 5px;
}

.fixed-header {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  background-color: white;
  padding: 10px;
  box-shadow: 0 2px 5px rgba(0,0,0,0.1);
  z-index: 100; /* 他の要素より手前に表示 */
}

.sticky-sidebar {
  position: sticky;
  top: 80px; /* fixed-header の高さ分くらい下で固定 */
}
```

開発者の視点: `position: absolute` や `fixed` は強力ですが、使いすぎるとレイアウトが崩れやすくなったり、管理が複雑になったりすることがあります。Flexbox や Grid で実現できるレイアウトであれば、そちらを優先する方が良い場合が多いです。`relative` は `absolute` の基準を作るためだけに `top` などを指定せずに使うこともよくあります。`sticky` は JavaScript を使わずに追従要素を実現できる便利な機能です。

4. レスポンシブデザイン：あらゆる画面サイズに対応する

現代では、パソコンだけでなく、スマートフォン、タブレットなど、様々な画面サイズのデバイスでWebサイトが閲覧されます。**レスポンシブデザイン** とは、これらの異なる画面サイズに応じて、Webページのレイアウトやデザインを自動的に調整する手法のことです。

レスポンシブデザインを実現するための主な技術は **メディアクエリ（Media Queries）** です。

メディアクエリ（`@media`）

メディアクエリを使うと、「画面幅が〇〇px以下の場合」や「画面が縦向きの場合」など、特定の条件を満たしたときにだけ適用されるCSSルールを書くことができます。

基本的な構文:

```
@media メディアタイプ and（条件式）{
  /* 条件に一致した場合に適用されるCSSルール */
  セレクタ {
    プロパティ: 値;
```

```
  }
}
```

- **メディアタイプ**: `all`（すべて）, `screen`（画面）, `print`（印刷）など。通常は `screen` を使うか、省略します（省略すると `all` 扱い）。
- **条件式**: `(max-width: 768px)`（画面幅が768px以下）, `(min-width: 769px)`（画面幅が769px以上）, `(orientation: portrait)`（縦向き）, `(orientation: landscape)`（横向き）など。`and` で複数の条件を組み合わせることもできます。

ブレークポイント: どの画面幅でレイアウトを切り替えるかの境界となる幅を **ブレークポイント** と呼びます。一般的なブレークポイントの例（プロジェクトによって異なります）：

- スマートフォン: 〜 600px
- タブレット: 601px 〜 900px
- デスクトップ: 901px 〜

モバイルファースト: レスポンシブデザインのアプローチとして、まずスマートフォン向けのスタイル（基本的なスタイル）を書き、その後メディアクエリを使って、より大きな画面向けのスタイルを上書きしていく「**モバイルファースト**」という考え方が推奨されています。

```
/* --- 基本スタイル（モバイルファースト）--- */
body {
  font-size: 16px;
  line-height: 1.6;
}

.container {
  width: 90%; /* モバイルでは幅を広めに */
  margin: 0 auto;
}

header nav ul {
  display: flex;
  flex-direction: column; /* モバイルでは縦並びメニュー */
  padding: 0;
  list-style: none;
}

header nav li {
  margin-bottom: 10px;
}

main {
  display: block; /* デフォルト */
}

aside {
  display: none; /* モバイルではサイドバー非表示 */
}

/* --- タブレット向けスタイル（例: 601px以上）--- */
```

```
@media screen and (min-width: 601px) {
  .container {
    width: 85%;
  }

  header nav ul {
    flex-direction: row; /* タブレットでは横並び */
    justify-content: space-around;
  }

  header nav li {
    margin-bottom: 0;
  }

  aside {
    display: block; /* タブレットではサイドバー表示 */
    float: right; /* 古典的な方法 or Grid/Flexで配置 */
    width: 30%;
    margin-left: 5%;
  }

  main {
     /* floatを使う場合 */
     /* float: left; */
     /* width: 65%; */
  }
  /* GridやFlexを使う方がモダン */
  /* .page-layout { display: grid; grid-template-columns: 2fr 1fr; gap: 20px; } */
  /* main { grid-column: 1; } */
  /* aside { grid-column: 2; } */
}

/* --- デスクトップ向けスタイル（例: 901px以上）--- */
@media screen and (min-width: 901px) {
  body {
    font-size: 18px; /* 少し文字を大きく */
  }
  .container {
    width: 80%;
    max-width: 1200px; /* 最大幅も設定 */
  }
  /* レイアウトはタブレットと同じか、さらに調整 */
}
```

ビューポートの設定

レスポンシブデザインを正しく機能させるためには、HTMLの `<head>` タグ内に以下の `<meta>` タグ（**ビューポートメタタグ**）を記述することがほぼ必須です。

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- `width=device-width`: ページの幅をデバイスの画面幅に合わせます。
- `initial-scale=1.0`: ページ読み込み時の初期ズーム倍率を1.0（等倍）にします。

これがないと、スマートフォンなどでPCサイトが縮小表示されてしまい、メディアクエリが意図した通りに動作しません。

開発者の視点: レスポンシブデザインは現代のWeb制作には不可欠です。モバイルファーストで進めると、コードがシンプルになりやすい傾向があります。ブラウザの開発者ツールには、様々なデバイスの画面サイズをシミュレートする機能があるので、必ずそれで表示を確認しながら開発を進めましょう。FlexboxやGridは、それ自体が柔軟なレイアウトシステムなので、レスポンシブデザインと非常に相性が良いです。

5. まとめ：自由自在なレイアウトを実現しよう

お疲れ様でした！これで、現代的なCSSレイアウト手法とレスポンシブデザインの基本を学ぶことができました。

- **Flexbox** は一次元（横一列 or 縦一列）のレイアウトを柔軟に制御するのに適しています。
- **Grid Layout** は二次元（行と列）の複雑なレイアウトを効率的に組むための強力なシステムです。
- `position` プロパティも、特定の要素を固定したり、重ねたりするのに役立ちます。
- **レスポンシブデザイン** と **メディアクエリ（`@media`）** を使うことで、あらゆる画面サイズのデバイスに対応したWebページを作成できます。
- **ビューポートメタタグ** はレスポンシブデザインの基本設定として重要です。

これらのレイアウト技術を理解し、使い分けることで、デザインカンプ（Webサイトの設計図）通りの見た目を効率的に実現したり、どんなデバイスで見ても使いやすいWebサイトを構築したりすることができるようになります。

レイアウトはCSSの中でも特に奥が深く、最初は難しく感じるかもしれませんが、実際にコードを書いて試してみるのが一番の近道です。ぜひ、色々なレイアウトパターンに挑戦してみてください！