

Canvas 実習マニュアル

第零版

2010 年 10 月 29 日 (金)

著者——大黒学

Copyright © 2010 Daikoku Manabu

This tutorial is licensed under a Creative Commons Attribution 2.1 Japan License.

目次

第 1 章 Canvas の基礎	4
1.1 Canvas の基礎の基礎	4
1.1.1 Canvas とは何か	4
1.1.2 このチュートリアルについて	4
1.1.3 canvas 要素	4
1.1.4 描画コンテキスト	5
1.1.5 Canvas の座標系	5
1.2 長方形	5
1.2.1 長方形を描画するメソッド	5
1.2.2 長方形の塗りつぶし	5
1.2.3 線による長方形の描画	6
1.2.4 グラフィックスの消去	6
1.2.5 要素オブジェクトの取得	7
1.3 色	7
1.3.1 スタイルのプロパティ	7
1.3.2 塗りつぶしの色	7
1.3.3 線の色	8
1.3.4 RGBA	8
1.3.5 アルファ値のプロパティ	9
1.4 画像データの URL	9
1.4.1 画像データの URL の取得	9
1.4.2 img 要素による画像データの表示	10
第 2 章 パス	10
2.1 パスの基礎	10
2.1.1 パスの基礎の基礎	10
2.1.2 カレントパスのリセット	11
2.1.3 サブパスの生成	11
2.1.4 直線	11
2.1.5 線によるカレントパスの描画	11
2.1.6 開いたサブパスと閉じたサブパス	12
2.1.7 カレントパスの塗りつぶし	12
2.1.8 クリッピング	13
2.2 ベジエ曲線	14
2.2.1 ベジエ曲線の基礎	14
2.2.2 二次ベジエ曲線	14
2.2.3 三次ベジエ曲線	14
2.3 円弧	15
2.3.1 円弧の基礎	15
2.3.2 arcTo	15
2.3.3 arc	16
2.3.4 扇形	16
2.4 線の形状	17

2.4.1	線の形状に影響を与えるプロパティ	17
2.4.2	端点の形状	17
2.4.3	接続点の形状	18
2.4.4	マイターリミット率	19
第 3 章	テキスト	19
3.1	テキストの基礎	20
3.1.1	テキストを描画するメソッド	20
3.1.2	フォント	20
3.1.3	テキストの輪郭を描画するメソッド	20
3.1.4	テキストの色	21
3.2	テキストの配置とベースラインと横幅	21
3.2.1	テキストの配置	21
3.2.2	テキストのベースライン	22
3.2.3	テキストの横幅	23
第 4 章	描画状態	24
4.1	描画状態の保存と復元	24
4.1.1	この章について	24
4.1.2	カレント描画状態	24
4.1.3	描画状態スタック	24
4.2	座標系の変換	25
4.2.1	座標系の変換の基礎	25
4.2.2	座標系の移動	25
4.2.3	拡大	26
4.2.4	座標系の回転	26
4.2.5	変換行列の直接的な変更	27
4.3	グラフィックスの装飾	27
4.3.1	グラフィックスの装飾の基礎	27
4.3.2	グラディエントの基礎	28
4.3.3	カラーストップ	28
4.3.4	線形グラディエント	28
4.3.5	放射状グラディエント	29
4.3.6	影	29
第 5 章	画像	30
5.1	画像の描画	30
5.1.1	この章について	30
5.1.2	Image オブジェクトの生成	30
5.1.3	Image オブジェクトの描画	30
5.1.4	Image オブジェクトを描画するタイミング	31
5.2	ピクセル操作	31
5.2.1	ピクセル操作の基礎	31
5.2.2	ImageData オブジェクトの取得	32
5.2.3	ImageData オブジェクトの描画	32
5.2.4	ImageData オブジェクトの生成	33
5.3	パターン	34
5.3.1	パターンの基礎	34
5.3.2	パターンをあらわすオブジェクト	34
第 6 章	Canvas の応用	34
6.1	アニメーション	34
6.1.1	setInterval	34
6.1.2	移動のアニメーション	35
6.2	イベント処理	36

目次	3
6.2.1 イベント属性	36
6.2.2 イベントオブジェクト	36
6.2.3 キーボード	37
6.3 フォーム要素との連携	37
6.3.1 テキストを入力するフォーム要素	37
6.3.2 設定された範囲内の数値を入力するフォーム要素	38
参考文献	39
索引	40

第1章 Canvasの基礎

1.1 Canvasの基礎の基礎

1.1.1 Canvasとは何か

HTML5は、`canvas`という要素型を持っています。この要素型の要素は、長方形の領域をウェブページの上に作ります。`canvas`要素によってウェブページの上に作られる長方形の領域は、「Canvas」と呼ばれます。

Canvasは、スクリプトを書くことによって、その上にグラフィックスを描画することができます、という機能を持っています。

1.1.2 このチュートリアルについて

このチュートリアルの目的は、Canvasの上にグラフィックスを描画する方法について解説することです。

このチュートリアルは、すでにHTMLとCSSとJavaScriptについての知識を持っている人を読者として想定して書かれています。ですから、それらについての知識をまだ持っていない人は、このチュートリアルを読む前に、あらかじめそれらについて勉強していただく必要があります。

1.1.3 canvas要素

`canvas`要素は、開始タグと終了タグを使って作ります。

`canvas`要素はインライン要素ですので、Canvasの前後のテキストやインライン要素は、Canvasの左右に表示されることになります。

Canvasの背景色やボーダーなどは、CSSを使うことによって指定することができます。デフォルトでは、背景色は透明で、ボーダーは非表示です。

Canvasの大きさは、次の属性を使うことによって指定することができます。

`width` Canvasの横の長さ。単位はピクセル。デフォルト値は300。

`height` Canvasの縦の長さ。単位はピクセル。デフォルト値は150。

たとえば、次のような`canvas`要素を書くことによって、`canvas1`という名前を持つ、横の長さが400ピクセルで縦の長さが300ピクセルのCanvasを作ることができます。

```
<canvas id="canvas1" width="400" height="300"></canvas>
```

次のHTML文書をブラウザで開くと、背景色とボーダーが異なる二つのCanvasが表示されるはずです。

HTML文書の例 `canvas.html`

```
<!DOCTYPE html>
<html>
<head>
<title>canvas</title>
<style type="text/css">
body {
    color: #000;
    background-color: #6f6;
}
#canvas1 {
    border: 5px solid #f00;
}
#canvas2 {
    color: #fff;
    background-color: #00f;
}
</style>
</head>
<body>
<canvas id="canvas1" width="200" height="150"></canvas>
<canvas id="canvas2" width="200" height="150"></canvas>
</body>
</html>
```

`width`と`height`の属性値は、スクリプトを使って変更することも可能ですが、変更すると、それ以前に描画されたグラフィックスは消去されます。

1.1.4 描画コンテキスト

`canvas`要素の要素オブジェクトは、「描画コンテキスト」(rendering context)と呼ばれるオブジェクトを持っています。`Canvas`の上にグラフィックスを描画するために必要となる各種の操作は、描画コンテキストが持っているメソッドを呼び出すことによって実行することができます。

`canvas`要素の要素オブジェクトから描画コンテキストを取得したいときは、`getContext`というメソッドを呼び出します。このメソッドには、「コンテキスト ID」(context ID)と呼ばれる文字列を引数として渡す必要があります。コンテキスト IDとして`2d`という文字列を渡すと、`getContext`は、`Canvas`の上に2次元のグラフィックスを描画するための描画コンテキストを戻り値として返します。

たとえば、次のような文を書くことによって、`canvas1`という名前を持つ`canvas`要素の要素オブジェクトを取得して、その要素オブジェクトから、`Canvas`の上に2次元のグラフィックスを描画するための描画コンテキストを取得することができます。

```
var context = document.getElementById("canvas1")
                .getContext("2d");
```

1.1.5 Canvasの座標系

`Canvas`の上での位置は、`Canvas`が持っている座標系を使うことによって指定することができます。

`Canvas`の座標系は、 x 軸と y 軸から構成されます。原点は`Canvas`の左上の隅にあって、 x 軸は右向き、 y 軸は下向きです。距離の単位はピクセルです。

1.2 長方形

1.2.1 長方形を描画するメソッド

描画コンテキストは、長方形を描画する次のようなメソッドを持っています。

`fillRect` 長方形の領域を塗りつぶす。

`strokeRect` 長方形を線で描画する。

`clearRect` 長方形の領域に描画されたグラフィックスを消去する。

これらの三つのメソッドは、すべて、長方形の位置と大きさを指定する、次の四つの引数を受け取ります。

1 個目 左上の頂点の x 座標。

2 個目 左上の頂点の y 座標。

3 個目 横の長さ。

4 個目 縦の長さ。

1.2.2 長方形の塗りつぶし

描画コンテキストが持っている `fillRect` というメソッドを呼び出すことによって、長方形の領域を塗りつぶすことができます。

HTML 文書の例 `fillrect.html`

```
<!DOCTYPE html>
<html>
<head>
<title>fillRect</title>
<style type="text/css">
#canvas1 {
    color: #fff;
    background-color: #6f6;
}
</style>
```

```

</head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.fillRect(100, 75, 200, 150);
</script>
</body>
</html>

```

1.2.3 線による長方形の描画

描画コンテキストが持っている `strokeRect` というメソッドを呼び出すことによって、線を使って長方形を描画することができます。

描画される線の太さは、描画コンテキストが持っている、`lineWidth` というプロパティに設定されている数値によって決定されます。太さの単位はピクセルで、デフォルトは 1 です。

HTML 文書の例 `strokeRect.html`

```

<!DOCTYPE html>
<html>
<head>
<title>strokeRect</title>
<style type="text/css">
#canvas1 {
    color: #fff;
    background-color: #6f6;
}
</style>
</head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.lineWidth = 20;
context.strokeRect(100, 75, 200, 150);
</script>
</body>
</html>

```

1.2.4 グラフィックスの消去

描画コンテキストが持っている `clearRect` というメソッドを呼び出すことによって、長方形の領域を指定して、その領域の中に描画されているグラフィックスを消去することができます。

HTML 文書の例 `clearRect.html`

```

<!DOCTYPE html>
<html>
<head>
<title>clearRect</title>
<style type="text/css">
#canvas1 {
    color: #fff;
    background-color: #6f6;
}
</style>
</head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.fillRect(20, 20, 360, 260);
context.clearRect(50, 50, 200, 150);
</script>

```

```
</body>
</html>
```

1.2.5 要素オブジェクトの取得

描画コンテキストは、`canvas` というプロパティを持っています。このプロパティには、`canvas` 要素の要素オブジェクトが設定されています。ですから、描画コンテキストから要素オブジェクトを取得することによって、`width` 属性や `height` 属性の値を取得したり、それらの属性値を変更したりする、ということが出来ます。

次の HTML 文書は、要素オブジェクトから `Canvas` の大きさを取得することによって、`Canvas` と同じ大きさの長方形を描画します。

HTML 文書の例 `sizeofcanvas.html`

```
<!DOCTYPE html>
<html>
<head><title>size of canvas</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

context.fillRect(
    0, 0, context.canvas.width, context.canvas.height);
</script>
</body>
</html>
```

1.3 色

1.3.1 スタイルのプロパティ

描画コンテキストは、グラフィックスを描画するときを使うスタイルが設定される、次の二つのプロパティを持っています。

`fillStyle` 塗りつぶしのスタイル。デフォルトは黒色。

`strokeStyle` 線のスタイル。デフォルトは黒色。

これらのプロパティにスタイルとして設定することができるのは、色、グラディエント、パターンのいずれかです。ただし、この節で説明するのは色を設定する方法だけです。グラディエントについては第 4.3 節で、パターンについては、第 5.3 節で説明することにしたいと思います。

1.3.2 塗りつぶしの色

`fillStyle` プロパティに色をあらわす文字列を設定しておく、グラフィックスを塗りつぶすときにその色が使われます。色をあらわす文字列の書き方は、CSS で定義されているものと同じです。

HTML 文書の例 `fillstyle.html`

```
<!DOCTYPE html>
<html>
<head><title>fillStyle</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillStyleRect(context, style, x, y) {
    context.fillStyle = style;
    context.fillRect(x, y, 200, 150);
}

var context = document.getElementById("canvas1")
                .getContext("2d");
fillStyleRect(context, "red", 20, 20);
fillStyleRect(context, "#0f0", 90, 70);
fillStyleRect(context, "rgb(0, 0, 255)", 160, 120);
```

```

</script>
</body>
</html>

```

1.3.3 線の色

strokeStyle プロパティに色をあらわす文字列を設定しておく、グラフィックスの線を描画するときその色が使われます。

HTML 文書の例 strokestyle.html

```

<!DOCTYPE html>
<html>
<head><title>strokeStyle</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function strokeStyleRect(context, style, x, y) {
    context.strokeStyle = style;
    context.strokeRect(x, y, 200, 150);
}

var context = document.getElementById("canvas1")
    .getContext("2d");
context.lineWidth = 20;
strokeStyleRect(context, "yellow", 20, 20);
strokeStyleRect(context, "#0ff", 90, 70);
strokeStyleRect(context, "rgb(255, 0, 255)", 160, 120);
</script>
</body>
</html>

```

1.3.4 RGBA

Canvas では、RGBA を使って色を指定することによって、透明なグラフィックスを描画することも可能です。

RGBA というのは、RGB にアルファ値を加えた四つの数値から構成される列のことです。RGBA を使って色をあらわす文字列は、

```
rgba(赤, 緑, 青, アルファ値)
```

と書きます。アルファ値は、0.0 から 1.0 までの数値で指定します。0.0 で完全に透明になって、1.0 で完全に不透明になります。

HTML 文書の例 rgba.html

```

<!DOCTYPE html>
<html>
<head><title>RGBA</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillAlphaRect(context, alpha, x) {
    context.fillStyle = "rgba(0, 0, 255, " + alpha + ")";
    context.fillRect(x, 10, 50, 280);
}

var context = document.getElementById("canvas1")
    .getContext("2d");
context.fillStyle = "lime";
context.fillRect(10, 50, 380, 200);
fillAlphaRect(context, 0.8, 70);
fillAlphaRect(context, 0.6, 140);
fillAlphaRect(context, 0.4, 210);
fillAlphaRect(context, 0.2, 280);
</script>
</body>
</html>

```

1.3.5 アルファ値のプロパティ

透明なグラフィックスを描画する方法は、もうひとつあります。それは、描画コンテキストが持っているプロパティにアルファ値を設定するという方法です。

描画コンテキストは、`globalAlpha` というプロパティを持っていて、デフォルトでは、完全な不透明を意味する `1.0` というアルファ値が設定されています。グラフィックスは、基本的には、このプロパティに設定されているアルファ値を持つ色で描画されます。

HTML 文書の例 `globalalpha.html`

```
<!DOCTYPE html>
<html>
<head><title>globalAlpha</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillStyleRect(context, style, x) {
    context.fillStyle = style;
    context.fillRect(x, 10, 50, 280);
}

var context = document.getElementById("canvas1")
    .getContext("2d");
context.fillStyle = "lime";
context.fillRect(10, 50, 380, 200);
context.globalAlpha = 0.6;
fillStyleRect(context, "darkorange", 70);
fillStyleRect(context, "deeppink", 140);
fillStyleRect(context, "skyblue", 210);
fillStyleRect(context, "indigo", 280);
</script>
</body>
</html>
```

1.4 画像データの URL

1.4.1 画像データの URL の取得

`canvas` 要素は、`toDataURL` というメソッドを持っています。このメソッドを呼び出すと、`Canvas` に描画されたグラフィックスの画像データを URL の形式にしたものが戻り値として返ってきます。

`toDataURL` には、引数として、取得する画像データの MIME タイプを渡します。引数を省略すると、`image/png` が指定されたと解釈されます。

`toDataURL` が返す URL は、

```
data: MIME タイプ;base64, 画像データを Base64 でエンコードした文字列
```

という形式になっています。

HTML 文書の例 `toDataURL.html`

```
<!DOCTYPE html>
<html>
<head><title>toDataURL</title></head>
<body>
<canvas id="canvas1" width="100" height="100"></canvas><br>
<span id="span1">hoge</span><br>
<a id="a1">開く</a>
<script type="text/javascript">
var canvas = document.getElementById("canvas1");
var context = canvas.getContext("2d");
context.fillStyle = "midnightblue";
context.fillRect(10, 10, 80, 80);
var url = canvas.toDataURL();
document.getElementById("span1").firstChild.data = url;
document.getElementById("a1").href = url;
```

```
</script>
</body>
</html>
```

1.4.2 img 要素による画像データの表示

toDataURL が返した URL を img 要素の src 属性に設定すると、その img 要素は、設定された URL によってあらわされる画像データを表示します。

HTML 文書の例 img.html

```
<!DOCTYPE html>
<html>
<head><title>img</title></head>
<body>
canvas<canvas id="canvas1" width="100" height="100"></canvas>
img<img id="img1">
<script type="text/javascript">
var canvas = document.getElementById("canvas1");
var context = canvas.getContext("2d");
context.fillStyle = "springgreen";
context.fillRect(10, 10, 80, 80);
document.getElementById("img1").src = canvas.toDataURL();
</script>
</body>
</html>
```

第2章 パス

2.1 パスの基礎

2.1.1 パスの基礎の基礎

第1.2節で説明したように、描画コンテキストは、長方形を描画するメソッドを持っています。ですから、Canvas に長方形を描画するために必要なことは、一つのメソッドを呼び出すことだけということになります。

しかし、一つのメソッドを呼び出すだけで描画することができる形状は、長方形だけです。長方形以外の形状を描画するためには、もう少し複雑な操作が必要になります。その操作は、大雑把に言うと、次の二段階のプロセスから構成されます。

- (1) パスを構築する。
- (2) そのパスを描画する。

「パス」(path) というのは、いくつかの線を組み合わせることによって作られた形状のことです。

一つのパスは、0 個または 1 個以上のサブパスから構成されます。「サブパス」(subpath) というのは、連続するいくつかの線を組み合わせることによって作られた形状のことです。パスを構築するというのは、1 個以上のサブパスを構築するということです。

描画コンテキストは、1 個のパスを保持することができます。描画コンテキストが保持しているパスは、「カレントパス」(current path) と呼ばれます。パスを描画するメソッドを呼び出すと、その時点でのカレントパスが描画されます。

パスの構築は、次の三段階のプロセスから構成されます。

- (1) カレントパスをリセットする。
- (2) サブパスを生成する。
- (3) 直線または曲線をサブパスに追加する。

サブパスの生成とサブパスへの線の追加は、何回でも繰り返すことができます。また、ひとつのサブパスに対する線の追加も、何回でも繰り返すことができます。

2.1.2 カレントパスのリセット

パスを構築したいときは、まず最初に、カレントパスをリセットするという操作を実行する必要があります。

すでにいくつかのサブパスが存在しているときにカレントパスをリセットした場合、それらのサブパスは破棄されて、カレントパスは、0個のサブパスから構成されている状態になります。

描画コンテキストが持っている `beginPath` というメソッドは、カレントパスをリセットします。たとえば、`context` が描画コンテキストだとするならば、

```
context.beginPath();
```

と書くことによって、カレントパスをリセットすることができます。

2.1.3 サブパスの生成

パスを構築するために次にしなければならないことは、空の新しいサブパスを生成して、カレントポイントを設定することです。

「カレントポイント」(current point) というのは、その言葉のとおり、「現在の点」という意味です。サブパスというのは、直線または曲線をサブパスに追加することによって構築するわけですが、それらの直線または曲線は、カレントポイントと次の点をつなぐことによって作られます。そして、次の点を指定することによって直線または曲線をサブパスに追加すると、カレントポイントは次の点に移動します。

描画コンテキストが持っている `moveTo` というメソッドは、空の新しいサブパスを生成して、引数として受け取った座標 (1 個目は x 座標、2 個目は y 座標) をカレントポイントとして設定します。たとえば、`context` が描画コンテキストだとするならば、

```
context.moveTo(70, 30);
```

と書くことによって、空の新しいサブパスを生成して、(70, 30) という座標をカレントポイントとして設定することができます。

2.1.4 直線

描画コンテキストが持っている `lineTo` というメソッドは、引数として次の点の座標を受け取って、カレントポイントと次の点をつなぐ直線を現在のサブパスに追加して、カレントポイントを次の点へ移動させます。たとえば、`context` が描画コンテキストだとするならば、

```
context.lineTo(200, 150);
```

と書くことによって、カレントポイントと (200, 150) とをつなぐ直線を現在のサブパスに追加して、カレントポイントを (200, 150) へ移動させることができます。

2.1.5 線によるカレントパスの描画

サブパスに直線や曲線を追加することによってパスを構築しても、その段階では、そのパスは Canvas の上に描画されません。パスを描画するためには、そのためのメソッドを呼び出す必要があります。

描画コンテキストが持っている `stroke` というメソッドは、線を使ってカレントパスを Canvas に描画します。たとえば、`context` が描画コンテキストだとするならば、

```
context.stroke();
```

と書くことによって、線を使ってカレントパスを Canvas に描画することができます。

線を使ってカレントパスを描画するとき、線の太さは、`lineWidth` プロパティによって決定されます。そして、線の色は、`strokeStyle` プロパティによって決定されます。

HTML 文書の例 path.html

```
<!DOCTYPE html>
<html>
<head><title>path</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.beginPath();
```

```

context.moveTo(50, 50);
context.lineTo(200, 50);
context.lineTo(200, 200);
context.lineTo(50, 200);
context.lineTo(50, 50);
context.lineWidth = 40;
context.strokeStyle = "crimson";
context.stroke();
</script>
</body>
</html>

```

2.1.6 開いたサブパスと閉じたサブパス

サブパスには、開いたものと閉じたものがあります。開いたサブパスというのは、始まりの点と終わりの点があるサブパスのことで、閉じたサブパスというのは、始まりの点も終わりの点もなく、その上をぐるりと一周することのできるサブパスのことで、

線を追加しただけのサブパスは、開いたサブパスです。始まりの点と終わりの点と同じ位置にあったとしても、それはやはり開いたサブパスです（先ほどの HTML 文書は、そのような例です）。閉じたサブパスを作るためには、サブパスを閉じるという動作をするメソッドを呼び出す必要があります。

描画コンテキストが持っている `closePath` というメソッドは、カレントポイントとサブパスの最初の点とを直線でつないだのち、サブパスを閉じます。

HTML 文書の例 `closepath.html`

```

<!DOCTYPE html>
<html>
<head><title>closePath</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(50, 50);
context.lineTo(200, 50);
context.lineTo(200, 200);
context.lineTo(50, 200);
context.closePath();
context.lineWidth = 40;
context.strokeStyle = "indigo";
context.stroke();
</script>
</body>
</html>

```

2.1.7 カレントパスの塗りつぶし

カレントパスは、線を使って描画することができるだけでなく、それによって囲まれた領域を塗りつぶすということも可能です。

描画コンテキストが持っている `fill` というメソッドは、カレントポイントとサブパスの最初の点とを直線でつないでサブパスを閉じたのち、カレントパスによって囲まれた領域を塗りつぶします。たとえば、`context` が描画コンテキストだとするならば、

```
context.stroke();
```

と書くことによって、線を使ってカレントパスを `Canvas` に描画することができます。

カレントパスによって囲まれた領域を塗りつぶすとき、その色は、`fillStyle` プロパティによって決定されます。

HTML 文書の例 `fill.html`

```

<!DOCTYPE html>
<html>
<head><title>fill</title></head>

```

```

<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(50, 50);
context.lineTo(200, 50);
context.lineTo(200, 200);
context.lineTo(50, 200);
context.fillStyle = "forestgreen";
context.fill();
</script>
</body>
</html>

```

2.1.8 クリッピング

ハサミを使って紙の一部を切り抜くように、グラフィックスの一部を切り抜いたグラフィックスを作ることを、「クリッピング」(clipping)と呼びます。

Canvas では、あらかじめグラフィックスを切り抜く領域を定義しておいて、そのうちグラフィックスを描画することによって、クリッピングを実現することができます。グラフィックスを切り抜く領域は、「クリッピング領域」(clipping region)と呼ばれます。

描画コンテキストは、「カレントクリッピング領域」(current clipping region)と呼ばれる領域を持っていて、グラフィックスをその領域でクリッピングします。デフォルトでは、Canvas と同じ位置と大きさを持つ長方形の領域がカレントクリッピング領域として設定されています。

カレントクリッピング領域は、描画コンテキストが持っている `clip` というメソッドを呼び出すことによって、より狭くすることができます。

`clip` は、カレントパスで囲まれた領域とカレントクリッピング領域との共通部分を、カレントクリッピング領域として設定します。カレントパスの中に開いたサブパスがある場合、クリッピング領域は、そのパスを仮に閉じたと考えて作られます。

HTML 文書の例 clip.html

```

<!DOCTYPE html>
<html>
<head><title>clip</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function striped(context) {
    context.beginPath();
    for (var y = 50; y <= 250; y+=10) {
        context.moveTo(50, y);
        context.lineTo(350, y);
    }
    context.stroke();
}

var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(140, 50);
context.lineTo(350, 150);
context.lineTo(140, 250);
context.clip();
context.beginPath();
context.moveTo(260, 50);
context.lineTo(50, 150);
context.lineTo(260, 250);
context.clip();
context.lineWidth = 4;
context.strokeStyle = "limegreen";
striped(context);
</script>

```

```
</body>
</html>
```

2.2 ベジエ曲線

2.2.1 ベジエ曲線の基礎

サブパスには、「ベジエ曲線」(Bézier curve) と呼ばれる曲線を追加することができます。ベジエ曲線の形状は、「制御点」(control point) と呼ばれる点の位置によって決定されます。

サブパスに追加することができるのは、次の 2 種類のベジエ曲線です。

- 二次ベジエ曲線 (quadratic Bézier curve)
- 三次ベジエ曲線 (cubic Bézier curve)

2.2.2 二次ベジエ曲線

二次ベジエ曲線の形状は、3 個の制御点によって決定されます。それぞれの制御点を、制御点 1、制御点 2、制御点 3 と呼ぶことにしましょう。二次ベジエ曲線は、制御点 1 から制御点 2 に向かって出発して、少しずつ方向を変えながら制御点 3 に到達する曲線です。

描画コンテキストが持っている `quadraticCurveTo` というメソッドは、二次ベジエ曲線をサブパスに追加します。このメソッドは、制御点 2 の x 座標と y 座標、制御点 3 の x 座標と y 座標を引数として受け取ります。制御点 1 はカレントポイントで、曲線をサブパスに追加したのち、カレントポイントは制御点 3 へ移動します。

HTML 文書の例 `quadratic.html`

```
<!DOCTYPE html>
<html>
<head><title>quadratic Bezier curve</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
    .getContext("2d");

context.beginPath();
context.moveTo(50, 250);
context.quadraticCurveTo(250, 50, 350, 250);
context.lineWidth = 40;
context.strokeStyle = "palegreen";
context.stroke();
context.beginPath();
context.moveTo(50, 250);
context.lineTo(250, 50);
context.lineTo(350, 250);
context.lineWidth = 2;
context.strokeStyle = "deeppink";
context.stroke();
</script>
</body>
</html>
```

2.2.3 三次ベジエ曲線

三次ベジエ曲線の形状は、4 個の制御点によって決定されます。それぞれの制御点を、制御点 1、制御点 2、制御点 3、制御点 4 と呼ぶことにしましょう。三次ベジエ曲線は、制御点 1 から制御点 2 に向かって出発して、少しずつ方向を変えながら、制御点 3 と制御点 4 とをつなぐ直線に接する形で、制御点 4 に到達する曲線です。

描画コンテキストが持っている `bezierCurveTo` というメソッドは、三次ベジエ曲線をサブパスに追加します。このメソッドは、制御点 2 の x 座標と y 座標、制御点 3 の x 座標と y 座標、制御点 4 の x 座標と y 座標を引数として受け取ります。制御点 1 はカレントポイントで、曲線をサブパスに追加したのち、カレントポイントは制御点 4 へ移動します。

HTML 文書の例 `cubic.html`

```
<!DOCTYPE html>
```

```

<html>
<head><title>cubic Bezier curve</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(100, 250);
context.bezierCurveTo(50, 100, 350, 50, 250, 250);
context.lineWidth = 40;
context.strokeStyle = "palegreen";
context.stroke();
context.beginPath();
context.moveTo(100, 250);
context.lineTo(50, 100);
context.moveTo(350, 50);
context.lineTo(250, 250);
context.lineWidth = 2;
context.strokeStyle = "deeppink";
context.stroke();
</script>
</body>
</html>

```

2.3 円弧

2.3.1 円弧の基礎

サブパスには、円弧を追加することもできます。「円弧」(arc)という言葉は、「円の一部分」という意味ですが、円の全体というのも円弧の特殊な場合だと考えることができます。

描画コンテキストは、サブパスに円弧を追加するメソッドとして、次の二つのものを持っています。

arcTo 2 個の点の座標と半径を指定して円弧を追加します。

arc 中心の座標、半径、開始角度、終了角度、回転の方向を指定して円弧を追加します。

2.3.2 arcTo

arcTo がサブパスに追加する円弧は、3 個の点と円弧の半径によって指定されます。円弧を指定するための 3 個の点のそれぞれを、点 1、点 2、点 3 と呼ぶことにしましょう。

arcTo は、引数として、点 2 の x 座標と y 座標、点 3 の x 座標と y 座標、そして円弧の半径を受け取ります。点 1 は、カレントポイントです。

点 1 と点 2 をつなぐ直線を接線 1、点 2 と点 3 をつなぐ直線を接線 2 と呼ぶことにしましょう。arcTo がサブパスに追加するのは、接線 1 と接線 2 の両方に接する円弧です。そこで、接線 1 と円弧とが接する点を接点 1、接線 2 と円弧とが接する点を接点 2 と呼ぶことにしましょう。

arcTo は、まず、点 1 (カレントポイント) と接点 1 とをつなぐ直線をサブパスに追加します。次に、接点 1 と接点 2 とをつなぐ円弧をサブパスに追加します。そして、カレントポイントを接点 2 へ移動させます。

HTML 文書の例 arcTo.html

```

<!DOCTYPE html>
<html>
<head><title>arcTo</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(50, 50);
context.arcTo(350, 50, 200, 250, 100);
context.lineWidth = 20;

```

```

context.strokeStyle = "skyblue";
context.stroke();
context.beginPath();
context.moveTo(50, 50);
context.lineTo(350, 50);
context.lineTo(200, 250);
context.lineWidth = 2;
context.strokeStyle = "crimson";
context.stroke();
</script>
</body>
</html>

```

2.3.3 arc

arc は、引数として、円弧の中心の x 座標と y 座標、半径、開始角度と終了角度、そして回転の方向を受け取ります。

arc は、moveTo と同じように、カレントパスが空の場合は、サブパスを生成します。サブパスが存在している場合は、カレントポイントと、円弧が開始する点とをつなぐ直線をサブパスに追加したのち、円弧をサブパスに追加します。そして、どちらの場合も、円弧が終了する点をカレントポイントとして設定します。

開始角度と終了角度は、右方向が 0 で、時計回りで大きくなります。角度の単位は、度ではなくてラジアンです。

回転の方向は、真偽値で指定します。偽を指定すると時計回り、真を指定すると反時計回りで円弧をサブパスに追加します。真を指定した場合も、開始角度と終了角度が大きくなる方向は、時計回りのままです。

HTML 文書の例 arc.html

```

<!DOCTYPE html>
<html>
<head><title>arc</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function toRadian(degree) {
    return Math.PI/180.0*degree;
}

var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.arc(300, 100, 70, toRadian(30), toRadian(150), false);
context.arc(100, 100, 70, toRadian(30), toRadian(150), true);
context.lineWidth = 10;
context.strokeStyle = "navy";
context.stroke();
</script>
</body>
</html>

```

2.3.4 扇形

次のような手順でサブパスを構築することによって、扇形を描画することができます。

- (1) moveTo を使って、円弧の中心をカレントパスとして設定する。
- (2) arc を使って、円弧をサブパスに追加する。
- (3) closePath を使って、サブパスを閉じる。

HTML 文書の例 sector.html

```

<!DOCTYPE html>
<html>
<head><title>sector</title></head>
<body>

```

```

<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function toRadian(degree) {
    return Math.PI/180.0*degree;
}

var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();
context.moveTo(200, 250);
context.arc(200, 250, 180, toRadian(210), toRadian(330), false);
context.closePath();
context.lineWidth = 20;
context.strokeStyle = "darkgreen";
context.stroke();
</script>
</body>
</html>

```

2.4 線の形状

2.4.1 線の形状に影響を与えるプロパティ

stroke または strokeRect を使って線を描画するとき、その線の形状は、描画コンテキストが持っている、次の四つのプロパティによる影響を受けます。

lineWidth 線の太さ。第 1.2.3 項参照。
lineCap 端点の形状。
lineJoin 接続点の形状。
miterLimit マイターリミット。

2.4.2 端点の形状

開いたサブパスが持っている始まりの点と終わり点のそれぞれは、「端点」(cap) と呼ばれます。端点の形状を指定したいときは、lineCap というプロパティに、その形状を意味する文字列を設定します。端点の形状を意味する文字列としては、次の三つのものがあります。

butt バットキャップ (butt cap)。端点を通る、線に対して垂直な直線で、線を切断した形状。デフォルトはこの形状。
round ラウンドキャップ (round cap)。端点を中心とする、線の太さを直径とする半円を付加した形状。
square プロジェクティングスクエアキャップ (projecting square cap)。バットキャップと同じように、線に対して垂直な直線で切断した形状ですが、線の太さの半分だけ線を延長したところで切断します。

HTML 文書の例 linecap.html

```

<!DOCTYPE html>
<html>
<head><title>lineCap</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function strokeLineCapPath(context, lineCap, y) {
    context.beginPath();
    context.moveTo(100, y);
    context.lineTo(300, y);
    context.lineCap = lineCap;
    context.stroke();
}

var context = document.getElementById("canvas1")
                .getContext("2d");

context.beginPath();

```

```

context.moveTo(100, 50);
context.lineTo(100, 250);
context.moveTo(300, 50);
context.lineTo(300, 250);
context.lineWidth = 2;
context.strokeStyle = "crimson";
context.stroke();
context.lineWidth = 40;
context.strokeStyle = "mediumblue";
strokeLineCapPath(context, "butt", 90);
strokeLineCapPath(context, "round", 150);
strokeLineCapPath(context, "square", 210);
</script>
</body>
</html>

```

2.4.3 接続点の形状

端点ではない点、つまり線と線とがそこで接続されている点は、「接続点」(join)と呼ばれます。

接続点の形状を指定したいときは、`lineJoin` というプロパティに、その形状を意味する文字列を設定します。端点の形状を意味する文字列としては、次の三つのものがあります。

- `miter` マイタージョイン (miter join)。線の外側の輪郭を、それらが交わるまで直線で延長した形状。デフォルトはこの形状。
- `round` ラウンドジョイン (round join)。線の外側の輪郭が、接続点を中心とする、線の太さを直径とする円を描く形状。
- `bevel` ベベルジョイン (bevel join)。両側の線を端点としてバットキャップで描画したときにできる三角形のくぼみを埋めた形状。

HTML 文書の例 `linejoin.html`

```

<!DOCTYPE html>
<html>
<head><title>lineJoin</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function strokeLineJoinPath(context, lineJoin, x) {
    context.beginPath();
    context.moveTo(x, 200);
    context.lineTo(x + 40, 100);
    context.lineTo(x + 80, 200);
    context.lineJoin = lineJoin;
    context.stroke();
}

var context = document.getElementById("canvas1")
                .getContext("2d");
context.lineWidth = 40;
context.strokeStyle = "maroon";
strokeLineJoinPath(context, "miter", 40);
strokeLineJoinPath(context, "round", 160);
strokeLineJoinPath(context, "bevel", 280);
</script>
</body>
</html>

```

`strokeRect` を使って長方形を描画する場合も、その線は、`lineJoin` プロパティによる影響を受けます。

HTML 文書の例 `rectline.html`

```

<!DOCTYPE html>
<html>
<head><title>lines of rectangle</title></head>
<body>

```

```

<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function strokeLineJoinRect(context, lineJoin, x, y) {
    context.lineJoin = lineJoin;
    context.strokeRect(x, y, 160, 120);
}

var context = document.getElementById("canvas1")
    .getContext("2d");
context.lineWidth = 40;
context.strokeStyle = "darkturquoise";
strokeLineJoinRect(context, "miter", 40, 40);
strokeLineJoinRect(context, "round", 110, 90);
strokeLineJoinRect(context, "bevel", 180, 140);
</script>
</body>
</html>

```

2.4.4 マイターリミット率

マイタージョインという接続点の形状は、線が十分に大きな角度で折れ曲がっている場合は問題がないのですが、きわめて小さな角度で折れ曲がっている場合、先端が長く伸びて不自然になります。ですから、Canvas では、接続点の形状としてマイタージョインが設定されていたとしても、角度が限界を超えて小さい場合には、ベベルジョインで接続点が描画されます。

接続点から、線の外側の輪郭が交わる点までの距離は、「マイター長」(miter length) と呼ばれます。そして、マイタージョインとベベルジョインのどちらになるかという境界となる、線の太さの半分に対するマイター長の比率は、「マイターリミット率」(miter limit ratio) と呼ばれます。

マイターリミット率を指定したいときは、`miterLimit` というプロパティにそれを設定します。

`miterLimit` には、デフォルト値として 10.0 が設定されています。この状態のとき、接続点の形状は、角度が約 11 度よりも小さくなったところで、マイタージョインからベベルジョインに切り替わります。

HTML 文書の例 `miterlimit.html`

```

<!DOCTYPE html>
<html>
<head><title>miterLimit</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function acute(context, y) {
    context.beginPath();
    context.moveTo(40, y);
    context.lineTo(250, y + 40);
    context.lineTo(40, y + 80);
    context.stroke();
}

var context = document.getElementById("canvas1")
    .getContext("2d");
context.lineWidth = 40;
context.strokeStyle = "mediumseagreen";
acute(context, 50);
context.miterLimit = 20;
acute(context, 180);
</script>
</body>
</html>

```

この HTML 文書をブラウザで開くと、同じ角度で折れ曲がる二つのパスが描画されます。上のパスは、マイターリミット率としてデフォルトの 10.0 が設定されている状態で描画されたもので、下のパスは、マイターリミット率として 20 が設定された状態で描画されたものです。

第3章 テキスト

3.1 テキストの基礎

3.1.1 テキストを描画するメソッド

描画コンテキストは、テキストを描画する、`fillText` というメソッドを持っています。

`fillText` は、次のような引数を受け取ります。

- 1 個目 描画するテキスト。
- 2 個目 テキストを描画する位置の x 座標。
- 3 個目 テキストを描画する位置の y 座標。
- 4 個目 テキストの横幅。

4 個目の引数は省略が可能です。この引数を渡すと、テキストは、指定された横幅の中に収まるように縮小されて描画されます。

HTML 文書の例 `filltext.html`

```
<!DOCTYPE html>
<html>
<head><title>fillText</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
    .getContext("2d");
context.fillText("私はテキストです。", 100, 50);
</script>
</body>
</html>
```

3.1.2 フォント

描画コンテキストは、`font` というプロパティを持っています。フォントをあらわす文字列をこのプロパティに設定すると、テキストを描画するときに、そのフォントが使われます。フォントをあらわす文字列の書き方は、CSS の `font` プロパティに設定する値の書き方と同じです。

HTML 文書の例 `font.html`

```
<!DOCTYPE html>
<html>
<head><title>font</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillFontText(context, font, y) {
    context.font = font;
    context.fillText(font, 50, y);
}

var context = document.getElementById("canvas1")
    .getContext("2d");
fillFontText(context, "30px serif", 80);
fillFontText(context, "30px monospace", 120);
fillFontText(context, "italic 30px serif", 160);
fillFontText(context, "bold 30px serif", 200);
fillFontText(context, "bold italic 30px serif", 240);
</script>
</body>
</html>
```

3.1.3 テキストの輪郭を描画するメソッド

描画コンテキストは、テキストの輪郭を描画する、`strokeText` というメソッドを持っています。このメソッドに渡す引数は、`fillText` に渡す引数と同じです。

テキストの輪郭となる線の太さは、線を使って長方形やパスを描画する場合と同じように、描画コンテキストが持っている `lineWidth` プロパティに設定されている数値によって決定されます。

HTML 文書の例 `strokertext.html`

```
<!DOCTYPE html>
<html>
<head><title>strokeText</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "240px serif";
context.lineWidth = 3;
context.strokeText("xyz", 50, 200);
</script>
</body>
</html>
```

3.1.4 テキストの色

テキストの描画に使われる色を設定する方法は、長方形やパスの色を設定する方法と同じです。

`fillText` を使ってテキストを描画する場合は、`fillStyle` プロパティに色を設定しておけば、その色が使われます。

同じように、`strokeText` を使ってテキストを描画する場合は、`strokeStyle` プロパティに色を設定しておけば、その色が使われます。

HTML 文書の例 `textcolor.html`

```
<!DOCTYPE html>
<html>
<head><title>text color</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "100px serif";
context.lineWidth = 2;
var color1 = "indigo";
var color2 = "crimson";
context.fillStyle = color1;
context.strokeStyle = color2;
context.fillText(color1, 50, 120);
context.strokeText(color2, 50, 250);
</script>
</body>
</html>
```

3.2 テキストの配置とベースラインと横幅

3.2.1 テキストの配置

テキストを描画するために指定した位置と、実際に描画されるテキストとの水平方向の位置関係は、そのテキストの「配置」(alignment)と呼ばれます。

描画コンテキストは、`textAlign` というプロパティを持っていて、テキストの配置は、このプロパティによって決定されます。

`textAlign` には、テキストの配置をあらわす、次の文字列のうちのいずれかを設定することができます。

- `start` 指定された位置をテキストの先頭にする。デフォルトはこの配置。
- `end` 指定された位置をテキストの末尾にする。
- `left` 左寄せ。
- `right` 右寄せ。
- `center` センタリング。

テキストを作るために文字を並べていく方向は、「書字方向」(text direction)と呼ばれます。CSSでは、デフォルトの書字方向は「左から右」に設定されていますが、

```
direction: rtl;
unicode-bidi: bidi-override;
```

と書くことによって、書字方向を「右から左」に変更することができます。

書字方向を逆にすると、textAlignに設定されたstartとendも、左寄せと右寄せとが逆になります。書字方向が「左から右」に設定されているときはstartが左寄せでendが右寄せ、書字方向が「右から左」に設定されているときはendが左寄せでstartが右寄せです。

HTML 文書の例 textalign.html

```
<!DOCTYPE html>
<html>
<head>
<title>textAlign</title>
<style type="text/css">
#canvas2 {
    direction: rtl;
    unicode-bidi: bidi-override;
}
</style>
</head>
<body>
<canvas id="canvas1" width="200" height="300"></canvas>
<canvas id="canvas2" width="200" height="300"></canvas>
<script type="text/javascript">
function fillAlignText(context, textAlign, y) {
    context.textAlign = textAlign;
    context.fillStyle = "darkgreen";
    context.font = "40px serif";
    context.fillText(textAlign, 100, y);
    context.beginPath();
    context.arc(100, y, 5, 0, Math.PI*2, false);
    context.fillStyle = "deeppink";
    context.fill();
}

function demoTextAlign(context) {
    fillAlignText(context, "start", 80);
    fillAlignText(context, "end", 120);
    fillAlignText(context, "left", 160);
    fillAlignText(context, "right", 200);
    fillAlignText(context, "center", 240);
}

demoTextAlign(document.getElementById("canvas1")
                .getContext("2d"));
demoTextAlign(document.getElementById("canvas2")
                .getContext("2d"));
</script>
</body>
</html>
```

3.2.2 テキストのベースライン

テキストを描画するために指定した位置と、実際に描画されるテキストとの垂直方向の位置関係は、そのテキストの「ベースライン」(baseline)と呼ばれます。

描画コンテキストは、textBaselineというプロパティを持っていて、テキストのベースラインは、このプロパティによって決定されます。

textBaselineには、テキストのベースラインをあらわす、

```
top hanging middle alphabetic ideographic bottom
```

という文字列のうちのいずれかを設定することができます(デフォルトはalphabeticです)。どの文字列を設定するとどのようなベースラインになるかということについては、次のHTML文

書をブラウザに表示させることによって確かめることができます。

HTML 文書の例 textbaseline.html

```
<!DOCTYPE html>
<html>
<head><title>textBaseline</title></head>
<body>
<canvas id="canvas1" width="650" height="200"></canvas>
<script type="text/javascript">
function fillBaselineText(context, textBaseline, x) {
    context.font = "18px serif";
    context.textBaseline = "alphabetic";
    context.fillText(textBaseline, x, 50);
    context.font = "40px serif";
    context.textBaseline = textBaseline;
    context.fillText("ky 私", x, 100);
}

var context = document.getElementById("canvas1")
                    .getContext("2d");

context.beginPath();
context.moveTo(0, 100);
context.lineTo(650, 100);
context.strokeStyle = "midnightblue";
context.lineWidth = 2;
context.stroke();
context.fillStyle = "maroon";
fillBaselineText(context, "top", 30);
fillBaselineText(context, "hanging", 130);
fillBaselineText(context, "middle", 230);
fillBaselineText(context, "alphabetic", 330);
fillBaselineText(context, "ideographic", 430);
fillBaselineText(context, "bottom", 530);
</script>
</body>
</html>
```

3.2.3 テキストの横幅

Canvas にグラフィックスを描画するとき、しばしば、設定されているフォントでテキストを描画した場合の横幅が必要になることがあります。

Canvas では、描画コンテキストが持っている `measureText` というメソッドを使うことによって、現在のフォントでテキストを描画した場合の横幅を取得することができます。

`measureText` は、引数としてテキストを受け取って、そのテキストを現在のフォントで描画した場合の横幅を持っているオブジェクトを戻り値として返します。そのオブジェクトは、`width` というプロパティを持っていて、描画されるテキストの横幅は、そのプロパティに設定されています。

次の HTML 文書は、テキストの横幅と同じ長さのアンダーラインを引くために `measureText` を使っています。

HTML 文書の例 underline.html

```
<!DOCTYPE html>
<html>
<head><title>underline</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillUnderlinedText(context, text, y) {
    context.fillText(text, 30, y);
    var width = context.measureText(text).width;
    context.beginPath();
    context.moveTo(30, y);
    context.lineTo(30+width, y);
    context.stroke();
}
</script>
```

```
var context = document.getElementById("canvas1")
                .getContext("2d");
context.lineWidth = 3;
context.font = "80px serif";
fillUnderlinedText(context, "onion", 80);
fillUnderlinedText(context, "eggplant", 170);
fillUnderlinedText(context, "lotus root", 260);
</script>
</body>
</html>
```

第4章 描画状態

4.1 描画状態の保存と復元

4.1.1 この章について

描画コンテキストは、グラフィックスの描画に関連するさまざまな状態を保持しています。たとえば、線の幅、線の形状、色、フォント、テキストの配置、テキストのベースライン、カレントクリッピング領域などです。それらの状態の集合は、「描画状態」(drawing state)と呼ばれます。ただし、カレントパスは、描画状態の中には含まれません。

この章では、描画状態を構成している状態のうちで、これまで紹介していなかったものを紹介していきたいと思います。

状態の紹介に先立って、この節では、描画状態のバックアップを作る方法と、バックアップしておいた描画状態を描画コンテキストに戻す方法について説明します。

4.1.2 カレント描画状態

描画コンテキストが持っている現在の描画状態は、「カレント描画状態」(current drawing state)と呼ばれます。

描画コンテキストは、カレント描画状態のバックアップを作るという機能と、そのバックアップを使ってカレント描画状態を以前の状態に復元するという機能を持っています。この機能を使うことによって、副作用を残すことなく描画状態を変更する関数を定義することができます。

4.1.3 描画状態スタック

描画コンテキストは、「描画状態スタック」(drawing state stack)と呼ばれるスタックを持っています。カレント描画状態のバックアップは、このスタックに保存されます。

描画コンテキストは、描画状態スタックを操作する、次の二つのメソッドを持っています。

`save` カレント描画状態を描画状態スタックにプッシュします。

`restore` 描画状態スタックから描画状態をポップして、それをカレント描画状態に復元します。

カレント描画状態を変更する関数の定義を、最初に `save` を呼び出して、最後に `restore` を呼び出すように書くと、その関数は、カレント描画状態の変更を副作用として残さないものになります。

HTML 文書の例 `save.html`

```
<!DOCTYPE html>
<html>
<head><title>save</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillBlueText(context, text, y) {
    context.save();
    context.fillStyle = "blue";
    context.fillText(text, 30, y);
    context.restore();
}
}
```

```
function fillRedText(context, text, y) {
    context.save();
    context.fillStyle = "red";
    context.fillText(text, 30, y);
    fillBlueText(context, text, y+40);
    context.fillText(text, 30, y+80);
    context.restore();
}

function fillNormalText(context, text, y) {
    context.fillText(text, 30, y);
    fillRedText(context, text, y+40);
    context.fillText(text, 30, y+160);
}

var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "30px serif";
context.fillStyle = "green";
fillNormalText(context, "The end justifies the means.", 50);
</script>
</body>
</html>
```

4.2 座標系の変換

4.2.1 座標系の変換の基礎

描画状態を構成している状態のひとつに、「変換行列」(transformation matrix) と呼ばれる行列があります。グラフィックスは、変換行列が適用された座標系を使って描画されます。デフォルトの描画状態が保持している変換行列は、変換を何もしない行列です。

描画コンテキストは、変換行列に変更を加えることによって座標系を変換する、次のようなメソッドを持っています。

translate	座標系を移動させます。
scale	座標系を拡大します。
rotate	座標系を回転させます。
transform	変換行列と、引数で指定された行列とを乗算した結果を変換行列にします。
setTransform	変換行列を初期化したのち、変換行列と、引数で指定された行列とを乗算した結果を変換行列にします。

4.2.2 座標系の移動

translate というメソッドを使うことによって、座標系を移動させることができます。このメソッドは、 x 軸方向と y 軸方向の移動量を引数として受け取ります。

HTML 文書の例 translate.html

```
<!DOCTYPE html>
<html>
<head><title>translate</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function circle(context, x, y, r) {
    context.save();
    context.translate(x, y);
    context.beginPath();
    context.arc(0, 0, r, 0, Math.PI*2, false);
    context.stroke();
    context.restore();
}

var context = document.getElementById("canvas1")
```

```

        .getContext("2d");
context.strokeStyle = "darkgreen";
context.lineWidth = 6;
circle(context, 70, 70, 20);
circle(context, 200, 150, 100);
circle(context, 350, 50, 200);
circle(context, 250, 500, 400);
</script>
</body>
</html>

```

4.2.3 拡大

scale というメソッドを使うことによって、座標系を拡大することができます。このメソッドは、 x 軸方向と y 軸方向の拡大率を引数として受け取ります。

HTML 文書の例 scale.html

```

<!DOCTYPE html>
<html>
<head><title>scale</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillScaleText(context, text, x, y, sx, sy) {
    context.save();
    context.translate(x, y);
    context.scale(sx, sy);
    context.fillText(text, 0, 0);
    context.restore();
}

var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "30px serif";
fillScaleText(context, "(1, 1)", 140, 120, 1, 1);
fillScaleText(context, "(2, 1)", 220, 120, 2, 1);
fillScaleText(context, "(1, 3)", 140, 220, 1, 3);
fillScaleText(context, "(-1, 1)", 120, 120, -1, 1);
fillScaleText(context, "(1, -1)", 140, 50, 1, -1);
</script>
</body>
</html>

```

4.2.4 座標系の回転

scale というメソッドを使うことによって、座標系を回転させることができます。このメソッドは、回転の角度（単位はラジアン）を引数として受け取ります。回転の中心は原点で、回転のプラスの方向は時計回りです。

HTML 文書の例 rotate.html

```

<!DOCTYPE html>
<html>
<head><title>rotate</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillRotateText(context, text, x, y, theta) {
    context.save();
    context.translate(x, y);
    context.rotate(theta);
    context.fillText(text, 0, 0);
    context.restore();
}

var context = document.getElementById("canvas1")
                .getContext("2d");

```

```

context.font = "40px serif";
fillRotateText(context, "0 radian", 50, 100, 0);
fillRotateText(context, "pi/4 radians", 200, 100, Math.PI/4);
fillRotateText(context, "pi radians", 250, 200, Math.PI);
</script>
</body>
</html>

```

4.2.5 変換行列の直接的な変更

transform または setTransform というメソッドを使うことによって、translate、scale、rotate にはできない変換を座標系に適用することができます。

transform は、変換行列と、引数で指定された行列とを乗算した結果を変換行列にします。

setTransform も、変換行列と、引数で指定された行列とを乗算した結果を変換行列にするのですが、transform とは違って、変換行列を初期化したのちに乗算をします。

これらの二つのメソッドは、引数として6個の数値を受け取ります。それらの数値を、順番に、 $m11$ 、 $m12$ 、 $m21$ 、 $m22$ 、 dx 、 dy と呼ぶことにすると、変換行列に乗算されるのは、次のような行列です。

$$\begin{pmatrix} m11 & m12 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{pmatrix}$$

次の HTML 文書は、setTransform を使って、座標軸を傾けるという変換（「剪断」(skew) と呼ばれます）を座標系に適用しています。

HTML 文書の例 settransform.html

```

<!DOCTYPE html>
<html>
<head><title>setTransform</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillSkewText(context, text, x, y, m12, m21) {
    context.save();
    context.setTransform(1, m12, m21, 1, x, y);
    context.fillText(text, 0, 0);
    context.restore();
}

var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "40px serif";
fillSkewText(context, "(0.4, 0)", 50, 100, 0.4, 0);
fillSkewText(context, "(-0.4, 0)", 200, 100, -0.4, 0);
fillSkewText(context, "(0, 0.4)", 50, 200, 0, 0.4);
fillSkewText(context, "(0, -0.4)", 200, 200, 0, -0.4);
</script>
</body>
</html>

```

4.3 グラフィックスの装飾

4.3.1 グラフィックスの装飾の基礎

描画状態を構成している状態の中には、グラフィックスの装飾に関連するものいくつかあって、それらの状態を変更することによって、グラフィックスに対して次のような装飾を与えることができます。

- グラディエント (gradient)
- パターン (pattern)
- 影 (shadow)

この節では、グラディエントと影について説明します。パターンについては、第5.3節で説明することにしたいと思います。

4.3.2 グラディエントの基礎

色を連続的に変化させるというグラフィックスのスタイルは、「グラディエント」(gradient)と呼ばれます。グラディエントには、次の2種類のものがあります。

- 線形グラディエント (linear gradient)
- 放射状グラディエント (radial gradient)

第1.3節で説明したように、色をあらわす文字列が `fillStyle` プロパティに設定されている場合は、グラフィックスを塗りつぶすときにその色が使われます。同じように、色をあらわす文字列が `strokeStyle` プロパティに設定されている場合は、線を描画するときにその色が使われます。

それらのプロパティに対して、色をあらわす文字列ではなくて、グラディエントをあらわすオブジェクトを設定する、ということも可能です。その場合は、グラフィックスを描画するときに、そのグラディエントがそのグラフィックスに適用されます。

4.3.3 カラーストップ

グラディエントをあらわすオブジェクトは、`addColorStop` というメソッドを持っています。これは、カラーストップをグラディエントに追加するメソッドです。

「カラーストップ」(color stop) というのは、色の変化を示す線の上の位置に対する色の設定のことです。カラーストップが設定されているそれぞれの位置のあいだは、色が連続的に変化することになります。

色の変化を示す線の上の位置は、0 から 1 までの数値によって示されます。0 はグラディエントを開始する位置、1 はグラディエントを終了する位置を示します。

たとえば、`grad` という変数にグラディエントのオブジェクトが設定されているとすると、

```
grad.addColorStop(0.6, "blue");
```

という文で `addColorStop` を呼び出すことによって、0.6 という位置に対して青色を設定するカラーストップをグラディエントに追加することができます。

4.3.4 線形グラディエント

描画コンテキストが持っている `createLinearGradient` というメソッドは、線形グラディエントをあらわすオブジェクトを生成して、それを戻り値として返します。

このメソッドは、4個の引数を受け取ります。1個目と2個目はグラディエントの開始点の x 座標と y 座標で、3個目と4個目はグラディエントの終了点の x 座標と y 座標です。

HTML 文書の例 `lineargradient.html`

```
<!DOCTYPE html>
<html>
<head><title>linear gradient</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillLinearGradientRect(context, x, y, x0, y0, x1, y1) {
    context.save();
    context.translate(x, y);
    var grad = context.createLinearGradient(x0, y0, x1, y1);
    grad.addColorStop(0, "aqua");
    grad.addColorStop(1, "blue");
    context.fillStyle = grad;
    context.fillRect(0, 0, 150, 100);
    context.restore();
}

var context = document.getElementById("canvas1")
    .getContext("2d");
fillLinearGradientRect(context, 50, 30, 0, 0, 150, 0);
fillLinearGradientRect(context, 230, 30, 0, 0, 0, 100);
fillLinearGradientRect(context, 50, 160, 0, 0, 150, 100);
```

```
fillLinearGradientRect(context, 230, 160, 150, 100, 0, 0);
</script>
</body>
</html>
```

4.3.5 放射状グラディエント

描画コンテキストが持っている `createRadialGradient` というメソッドは、放射状グラディエントをあらわすオブジェクトを生成して、それを戻り値として返します。

このメソッドは、6 個の引数を受け取ります。1 個目から 3 個目までは、グラディエントを開始する円の中心の x 座標と y 座標と半径で、4 個目から 6 個目までは、グラディエントを終了する円の中心の x 座標と y 座標と半径です。

HTML 文書の例 `radialgradient.html`

```
<!DOCTYPE html>
<html>
<head><title>radial gradient</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillRadialGradientArc(context, x, y, r, color) {
    context.save();
    var grad = context.createRadialGradient(
        x-r*0.3, y-r*0.3, 0, x-r*0.6, y-r*0.6, r*1.8);
    grad.addColorStop(0, "white");
    grad.addColorStop(1, color);
    context.fillStyle = grad;
    context.beginPath();
    context.arc(x, y, r, 0, Math.PI*2, false);
    context.fill();
    context.restore();
}

var context = document.getElementById("canvas1")
    .getContext("2d");
fillRadialGradientArc(context, 150, 150, 100, "brown");
fillRadialGradientArc(context, 300, 200, 50, "midnightblue");
</script>
</body>
</html>
```

4.3.6 影

描画コンテキストが持っている次のプロパティに値を設定することによって、グラフィックスに影をつけることができます。

`shadowOffsetX` x 軸方向の影の位置。単位はピクセル。デフォルトは 0。
`shadowOffsetY` y 軸方向の影の位置。単位はピクセル。デフォルトは 0。
`shadowBlur` 影のぼかし度。デフォルトは 0。
`shadowColor` 影の色。デフォルトは透明な黒色。

HTML 文書の例 `shadow.html`

```
<!DOCTYPE html>
<html>
<head><title>shadow</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function fillShadowRect(context, x, y, blur, color) {
    context.save();
    context.fillStyle = "green";
    context.shadowOffsetX = 15;
    context.shadowOffsetY = 10;
    context.shadowBlur = blur;

```

```
    context.shadowColor = color;
    context.fillRect(x, y, 150, 100);
    context.restore();
}

var context = document.getElementById("canvas1")
                .getContext("2d");
context.font = "30px serif";
fillShadowRect(context, 40, 30, 0, "orange");
fillShadowRect(context, 220, 30, 0, "gray");
fillShadowRect(context, 40, 160, 4, "gray");
fillShadowRect(context, 220, 160, 8, "gray");
</script>
</body>
</html>
```

第5章 画像

5.1 画像の描画

5.1.1 この章について

これまで、Canvas の上に図形やテキストなどのグラフィックスを描画する方法について説明してきましたが、Canvas の上に描画することができるグラフィックスは、図形やテキストだけではありません。画像を Canvas の上に描画するというのも可能です。

さらに、Canvas の上に描画されたグラフィックスは、ビットマップの画像データとして取得することができて、その画像データを加工して、その結果を Canvas に描画する、ということも可能です。

そこで、この章では、Canvas が持っている機能のうちで、画像に関連するものについて説明していきたいと思います。

5.1.2 Image オブジェクトの生成

この節では、URL で指定された画像データをロードして、それを Canvas の上に描画する方法について説明したいと思います。

URL で指定された画像データを Canvas の上に描画するためには、その画像データを保持する Image オブジェクトを生成する必要があります。たとえば、

```
var image = new Image();
image.src = "sample.jpg";
```

と書くことによって、HTML 文書と同じディレクトリにある sample.jpg というファイルからロードされた画像データを保持する Image オブジェクトを作ることができます。

5.1.3 Image オブジェクトの描画

描画コンテキストが持っている drawImage というメソッドを使うことによって、Image オブジェクトが保持している画像データを Canvas の上に描画することができます。

drawImage に渡す引数の個数は、3 個、5 個、9 個のいずれかです。いずれの場合も、1 個目の引数は、画像データを保持している、Image オブジェクトなどのオブジェクトです。

3 個の引数を渡す場合、2 個目と 3 個目は画像を表示する位置の左上の x 座標と y 座標です。

5 個の引数を渡すことによって、描画する画像の大きさを指定することができます。その場合、2 個目と 3 個目は画像を表示する位置の左上の x 座標と y 座標、4 個目と 5 個目は画像の横の長さ、5 個目と 6 個目は縦の長さです。

9 個の引数を渡すことによって、画像の一部分を長方形で切り取って、その部分だけを描画する、ということができます。その場合、2 個目と 3 個目は画像を切り取る長方形の左上の x 座標と y 座標、4 個目と 5 個目はその長方形の横の長さ、6 個目と 7 個目は画像を表示する位置の左上の x 座標と y 座標、8 個目と 9 個目は画像の横の長さ、9 個目は縦の長さです。

5.1.4 Image オブジェクトを描画するタイミング

Image オブジェクトが生成されてから画像データのロードが完了するまでには、多少の時間がかかります。ですから、

```
var image = new Image();
image.src = "sample.jpg";
context.drawImage(image, 50, 50, 300, 200);
```

というスクリプトでは、画像データのロードが完了する以前の段階で drawImage が呼び出されるために、画像は描画されません。

画像を描画するスクリプトは、画像データのロードが完了したのちに drawImage が呼び出されるように、

```
var image = new Image();
image.src = "sample.jpg";
image.onload = function() {
    context.drawImage(image, 50, 50, 300, 200);
};
```

このように書く必要があります。

HTML 文書の例 drawimage.html

```
<!DOCTYPE html>
<html>
<head><title>drawImage</title></head>
<body>
<canvas id="canvas1" width="200" height="300"></canvas>
<canvas id="canvas2" width="200" height="300"></canvas>
<canvas id="canvas3" width="200" height="300"></canvas>
<script type="text/javascript">
function getContext(canvas) {
    return document.getElementById(canvas).getContext("2d");
}

var image = new Image();
image.src = "sample.jpg";
image.onload = function() {
    getContext("canvas1").drawImage(image, 0, 0);
    getContext("canvas2").drawImage(image, 0, 0, 200, 300);
    getContext("canvas3").drawImage(image, 0, 0,
        image.naturalWidth/2, image.naturalHeight/2,
        0, 0, 200, 300);
};
</script>
</body>
</html>
```

5.2 ピクセル操作

5.2.1 ピクセル操作の基礎

ピクセルのデータを操作する処理は、「ピクセル操作」(pixel manipulation) と呼ばれます。

Canvas でピクセル操作をするためには、「ImageData オブジェクト」(ImageData object) と呼ばれるオブジェクトが必要になります。このオブジェクトは、「ビットマップ」(bitmap) と呼ばれるものを持っています。ビットマップというのは、個々のピクセルのデータから構成される配列のことです。

描画コンテキストは、ImageData オブジェクトを扱う、次のようなメソッドを持っています。

getImageData	Canvas に描画されているグラフィックスのビットマップを持っている ImageData オブジェクトを取得します。
putImageData	ImageData オブジェクトが持っているビットマップを Canvas に描画します。
createImageData	新しいビットマップを持っている ImageData オブジェクトを生成します。

5.2.2 ImageData オブジェクトの取得

描画コンテキストが持っている `getImageData` というメソッドは、Canvas に描画されているグラフィックスのビットマップを持っている `ImageData` オブジェクトを取得します。このメソッドは、4 個の引数を受け取ります。1 個目と 2 個目は、取得する範囲を示す長方形の左上の x 座標と y 座標で、3 個目と 4 個目は、その長方形の横の長さで縦の長さです。

`ImageData` オブジェクトは、次のようなプロパティから構成されています。

`width` ビットマップがあらわしている画像の横の長さ。

`height` ビットマップがあらわしている画像の縦の長さ。

`data` ビットマップ。

`data` プロパティが持っているビットマップは、整数の 1 次元配列です。個々のピクセルのデータは、赤、緑、青、アルファ値を示す 4 個の整数 (0 から 255 まで) から構成されています。そして、それらのピクセルのデータは、先頭が画像の左上の隅で、そこから右に向かって進んでいって、右端に到達すると 1 ピクセル下の段の左端へ行く、という順序で並んでいます。

次の HTML 文書は、縦が 2 ピクセル、横も 2 ピクセルの Canvas から取得したビットマップを数字で表示します。

HTML 文書の例 `getImageData.html`

```
<!DOCTYPE html>
<html>
<head><title>getImageData</title></head>
<body id="body">
<canvas id="canvas1" width="2" height="2"></canvas><br>
<span id="span1">hoge</span>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.fillStyle = "rgba(32, 128, 64, 1.0)";
context.fillRect(0, 0, 1, 1);
context.fillStyle = "rgba(128, 64, 32, 1.0)";
context.fillRect(1, 1, 1, 1);
var image = context.getImageData(0, 0, 2, 2);
var s = "";
for (var i = 0; i < image.data.length; i++)
    s += image.data[i] + " ";
document.getElementById("span1").firstChild.data = s;
</script>
</body>
</html>
```

5.2.3 ImageData オブジェクトの描画

描画コンテキストが持っている `putImageData` というメソッドは、`ImageData` オブジェクトが持っているビットマップを Canvas に描画します。このメソッドは、3 個の引数を受け取ります (7 個の引数を渡すことによって、ビットマップの一部分だけを描画することも可能です)。1 個目は `ImageData` オブジェクトで、2 個目と 3 個目は、描画する位置を示す x 座標と y 座標です。

次の HTML 文書は、`getImageData` と `putImageData` を使うことによって、Canvas に描画されたグラフィックスの色を補色に変換しています。

HTML 文書の例 `putImageData.html`

```
<!DOCTYPE html>
<html>
<head><title>putImageData</title></head>
<body id="body">
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
function concentricCircles(context, x, y, step, color) {
    context.save();
    context.lineWidth = 8;
    context.strokeStyle = color;
    context.translate(x, y);
    for (var r = 10; r < 500; r += step) {
```

```

        context.beginPath();
        context.arc(0, 0, r, 0, Math.PI*2, false);
        context.stroke();
    }
    context.restore();
}

function complementaryColor(context, x, y, width, height) {
    var image = context.getImageData(x, y, width, height);
    for (var i = 0; i < image.data.length; i++)
        if (i%4 != 3)
            image.data[i] = 255 - image.data[i];
    context.putImageData(image, x, y);
}

var context = document.getElementById("canvas1")
                .getContext("2d");
concentricCircles(context, 50, 120, 20, "orange");
concentricCircles(context, 340, 200, 25, "springgreen");
concentricCircles(context, 380, 30, 30, "hotpink");
complementaryColor(context, 50, 50, 250, 150);
complementaryColor(context, 200, 100, 150, 150);
</script>
</body>
</html>

```

5.2.4 ImageData オブジェクトの生成

描画コンテキストが持っている `createImageData` というメソッドは、`ImageData` オブジェクトを生成します。このメソッドは、引数として、ビットマップがあらわす画像の大きさを受け取ります。

画像の大きさを指定する方法は、二つあります。ひとつは、横の長さと同縦の長さを示す 2 個の整数を引数として渡すという方法で、もうひとつは、生成してほしいビットマップと同じ大きさのビットマップを持つ `ImageData` オブジェクトを引数として渡すという方法です。

`createImageData` によって生成された直後の `ImageData` オブジェクトが持っているビットマップは、すべてのピクセルが透明な黒色に初期化されています。

次の HTML 文書は、`createImageData` と `putImageData` を使うことによって、左上の隅が黒色で、右へ行くほど緑色の成分が明るくなって、下へ行くほど青色の成分が明るくなるような正方形を描画しています。

HTML 文書の例 `createImageData.html`

```

<!DOCTYPE html>
<html>
<head><title>createImageData</title></head>
<body id="body">
<canvas id="canvas1" width="256" height="256"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");

var image = context.createImageData(256, 256);
for (var b = 0; b < 256; b++)
    for (var g = 0; g < 256; g++) {
        var i = b*1024 + g*4;
        image.data[i+1] = g;
        image.data[i+2] = b;
        image.data[i+3] = 255;
    }
context.putImageData(image, 0, 0);
</script>
</body>
</html>

```

5.3 パターン

5.3.1 パターンの基礎

第 1.3 節で説明したように、`fillStyle` と `strokeStyle` というプロパティには、色、グラディエント、パターンのいずれかを設定することができます。

`fillStyle` プロパティにパターンを設定しておいて、その状態でグラフィックスの内部を塗りつぶすと、そのグラフィックスは、設定されているパターンで塗りつぶされることとなります。同じように、`strokeStyle` プロパティにパターンを設定しておいて、その状態で線を描画すると、その線は、設定されているパターンで描画されることとなります。

5.3.2 パターンをあらわすオブジェクト

`fillStyle` または `strokeStyle` プロパティにパターンを設定するためには、そのパターンをあらわすオブジェクトを作って、そのオブジェクトをそれらのプロパティに設定する必要があります。

パターンをあらわすオブジェクトは、描画コンテキストが持っている `createPattern` というメソッドを呼び出すことによって生成することができます。このメソッドには、2 個の引数を渡します。

1 個目の引数は、パターンになる画像をあらわすオブジェクトです。第 5.1 節で説明した `Image` オブジェクトを 1 個目の引数として渡すと、その `Image` オブジェクトが保持している画像データによるパターンのオブジェクトが生成されます。

2 個目の引数は、パターンの繰り返しを指定する文字列です。繰り返しは、次のような文字列によって指定されます。

```
repeat      x 軸と y 軸の両方向に繰り返す。
repeat-x    x 軸方向に繰り返す。
repeat-y    y 軸方向に繰り返す。
no-repeat   繰り返さない。
```

HTML 文書の例 pattern.html

```
<!DOCTYPE html>
<html>
<head><title>pattern</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
var image = new Image();
image.src = "sample.png";
image.onload = function() {
    context.fillStyle = context.createPattern(image, "repeat");
    context.fillRect(0, 0, 400, 300);
};
</script>
</body>
</html>
```

第 6 章 Canvas の応用

6.1 アニメーション

6.1.1 setInterval

Canvas に描画することのできるグラフィックスは、静止したものだけではありません。Canvas を使ってアニメーションを作るということも可能です。

Canvas でアニメーションを作りたいときは、`setInterval` というメソッドを使います。このメソッドは、引数として関数と時間（単位はミリ秒）を受け取って、その時間ごとにその関数を

呼び出します。

HTML 文書の例 expand.html

```
<!DOCTYPE html>
<html>
<head><title>expand</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.fillStyle = "cornflowerblue";
var r = 1;
setInterval(function() { fillCircle(context, r++);}, 100);

function fillCircle(context, r) {
    context.beginPath();
    context.arc(200, 150, r, 0, Math.PI*2, false);
    context.fill();
}
</script>
</body>
</html>
```

6.1.2 移動のアニメーション

グラフィックスを移動させるアニメーションを作るためには、グラフィックスを消去するという処理が必要になります。なぜなら、移動前のグラフィックスを消去しないで移動後のグラフィックスを描画すると、移動の軌跡が Canvas の上に残ってしまうからです。

グラフィックスを消去したいときは、第 1.2 節で紹介した `clearRect` というメソッドを使います。

HTML 文書の例 roundtrip.html

```
<!DOCTYPE html>
<html>
<head><title>round-trip</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
                .getContext("2d");
context.fillStyle = "darkmagenta";
var x = 5;
var direction = 1;
setInterval(function() {
    if (direction == 1) {
        if (x >= 380)
            direction = 2;
        else
            x += 5;
    } else {
        if (x <= 20)
            direction = 1;
        else
            x -= 5;
    }
    context.clearRect(0, 0, 400, 300);
    fillCircle(context, x);
}, 100);

function fillCircle(context, x) {
    context.save();
    context.translate(x, 150);
    context.beginPath();
    context.arc(0, 0, 20, 0, Math.PI*2, false);
    context.fill();
}
```

```
    context.restore();
}
</script>
</body>
</html>
```

6.2 イベント処理

6.2.1 イベント属性

HTML の要素の多くは、「イベント属性」(event attribute) と呼ばれるいくつかの属性を持っています。要素が持っているイベント属性にスクリプトを設定しておく、その要素でイベントが発生したときに、そのスクリプトが実行されます。

たとえば、`onclick` というイベント属性に設定されたスクリプトは、要素がクリックされたときに実行されます。

`canvas` 要素もイベント属性を持っていますので、それらを使うことによって、イベントが発生したときに何らかの変化が生じる、インタラクティブなグラフィックスを作ることができます。

HTML 文書の例 `onclick.html`

```
<!DOCTYPE html>
<html>
<head><title>onclick</title></head>
<body>
<canvas id="canvas1" width="400" height="300"
  onclick="incrementCount()"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
  .getContext("2d");
context.fillStyle = "coral";
var count = 0;
context.font = "200px serif";
incrementCount();

function incrementCount() {
  context.clearRect(0, 0, 400, 300);
  context.fillText(count++, 50, 200);
}
</script>
</body>
</html>
```

6.2.2 イベントオブジェクト

イベントが発生すると、「イベントオブジェクト」(event object) と呼ばれる、発生したイベントについての情報を持つオブジェクトが生成されます。

イベントオブジェクトは、スクリプトの中に `event` と書くことによって参照することができます。

イベントオブジェクトはさまざまなプロパティを持っていて、発生したイベントについての情報が、それぞれのプロパティに設定されています。たとえば、`offsetX` と `offsetY` というプロパティには、要素の左上を原点とする座標系で、マウスポインターの位置が設定されています。

HTML 文書の例 `position.html`

```
<!DOCTYPE html>
<html>
<head><title>position</title></head>
<body>
<canvas id="canvas1" width="400" height="300"
  onclick="clickHandler(event)"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
  .getContext("2d");
context.fillStyle = "deeppink";
```

```
function fillCircle(context, x, y) {
    context.save();
    context.translate(x, y);
    context.beginPath();
    context.arc(0, 0, 10, 0, Math.PI*2, false);
    context.fill();
    context.restore();
}

function clickHandler(event) {
    fillCircle(context, event.offsetX, event.offsetY);
}
</script>
</body>
</html>
```

6.2.3 キーボード

HTML 文書そのものをあらわしている `document` というオブジェクトは、`onkeydown` というプロパティを持っています。このプロパティに関数を設定しておく、その関数は、キーボードのキーが押されたときに呼び出されます。

`onkeydown` に設定する関数に 1 個の仮引数を宣言しておく、その仮引数にはイベントオブジェクトが設定されます。

キーボードを構成しているそれぞれのキーは、識別のために、「キーコード」(key code) と呼ばれる整数を持っています。イベントオブジェクトが持っている `keyCode` というプロパティには、押されたキーのキーコードが設定されます。ですから、このプロパティを使うことによって、キーボードのどのキーが押されたのかということを知ることができます。

次の HTML 文書は、キーボードのキーが押されたときに、押されたキーのキーコードを表示します。

HTML 文書の例 `onkeydown.html`

```
<!DOCTYPE html>
<html>
<head><title>onkeydown</title></head>
<body>
<canvas id="canvas1" width="400" height="300"></canvas>
<script type="text/javascript">
var context = document.getElementById("canvas1")
    .getContext("2d");
context.font = "200px serif";
context.fillStyle = "royalblue";
document.onkeydown = function(event) {
    context.clearRect(0, 0, 400, 300);
    context.fillText(event.keyCode, 50, 200);
};
</script>
</body>
</html>
```

6.3 フォーム要素との連携

6.3.1 テキストを入力するフォーム要素

この節では、Canvas 要素とフォーム要素との連携について説明したいと思います。

まず最初は、テキストを入力するフォーム要素と Canvas 要素との連携です。

テキストを入力するフォーム要素は、`input` 要素の `type` 属性に対して、`text` という属性値を設定することによって作ることができます。

`text` タイプのフォーム要素に入力されているテキストに変更が加わると、`oninput` というイベント属性に設定されているスクリプトが実行されます。

イベントオブジェクトは、`target` というプロパティを持っています。このプロパティには、イベントが発生した要素の要素オブジェクトが設定されています。`text` タイプのフォーム要素の

場合、要素オブジェクトが持っている value 属性から、入力されているテキストを取得することができます。

HTML 文書の例 inputtext.html

```
<!DOCTYPE html>
<html>
<head><title>input text</title></head>
<body>
<canvas id="canvas1" width="600" height="100"></canvas><br>
<input type="text" size="30" autofocus
  oninput="changeText(event)">
<script type="text/javascript">
var context = document.getElementById("canvas1")
  .getContext("2d");
context.font = "40px serif";
context.fillStyle = "indigo";

function changeText(event) {
  context.clearRect(0, 0, 600, 100);
  context.fillText(event.target.value, 10, 80);
}
</script>
</body>
</html>
```

6.3.2 設定された範囲内の数値を入力するフォーム要素

HTML5 では、input 要素の type 属性に対して、range という属性値を設定することができます。range を type 属性に設定した場合、その input 要素は、設定された範囲内の数値を入力することのできるフォーム要素になります。

range タイプの input 要素に入力することのできる数値の範囲と増減のステップは、次の属性を使って設定します。

min 最小値。デフォルトは 0。
max 最大値。デフォルトは 100。
step 増減のステップ。デフォルトは 1。

HTML 文書の例 inputrange.html

```
<!DOCTYPE html>
<html>
<head>
<title>input range</title>
<style type="text/css">
input { width: 300px; }
</style>
</head>
<body>
<canvas id="canvas1" width="300" height="300"></canvas><br>
<input type="range" min="0" max="150" step="1" value="50"
  onchange="changeRadius(event)">
<script type="text/javascript">
var context = document.getElementById("canvas1")
  .getContext("2d");
context.fillStyle = "lawngreen";
fillCircle(context, 50);

function fillCircle(context, r) {
  context.beginPath();
  context.arc(150, 150, r, 0, Math.PI*2, false);
  context.fill();
}

function changeRadius(event) {
  context.clearRect(0, 0, 300, 300);
  fillCircle(context, event.target.value);
}
```

```
}  
</script>  
</body>  
</html>
```

参考文献

- [HTML5,2010] Ian Hickson ed., “HTML5: A vocabulary and associated APIs for HTML and XHTML”, World Wide Web Consortium, 2010.
- [HTMLCanvas2DContext,2010] Ian Hickson ed., “HTML Canvas 2D Context”, World Wide Web Consortium, 2010.
- [白石,2010] 白石俊平、『HTML5&API 入門』、日経 BP 社、2010、ISBN 978-4-8222-8422-0。
- [羽田野,2010] 羽田野太巳、白石俊平、古籟一浩、太田昌吾、『Google API Expert が解説する HTML5 ガイドブック』、インプレスジャパン、2010、ISBN 978-4-8443-2927-5。

索引

- step 属性, 38
- 2d, 5

- addColorStop メソッド, 28
- alphabetic (テキストのベースライン), 22
- arc メソッド, 15, 16
- arcTo メソッド, 15

- beginPath メソッド, 11
- bevel (接続点の形状), 18
- bezierCurveTo メソッド, 14
- bottom (テキストのベースライン), 22
- butt (端点の形状), 17

- Canvas, 4
 - の大きさ, 4, 7
 - の座標系, 5
- canvas プロパティ, 7
- canvas 要素, 4, 7, 9, 36
- center (テキストの配置), 21
- clearRect メソッド, 5, 6, 35
- clip メソッド, 13
- closePath メソッド, 12
- createImageData メソッド, 31, 33
- createLinearGradient メソッド, 28
- createPattern メソッド, 34
- createRadialGradient メソッド, 29
- CSS, 4, 7, 20

- data プロパティ, 32
- document, 37
- drawImage メソッド, 30

- end (テキストの配置), 21
- event, 36

- fill メソッド, 12
- fillRect メソッド, 5
- fillStyle プロパティ, 7, 12, 21, 28, 34
- fillText メソッド, 20
- font プロパティ, 20

- getContext メソッド, 5
- getImageData メソッド, 31, 32
- globalAlpha プロパティ, 9

- hanging (テキストのベースライン), 22
- height プロパティ, 32
- height 属性, 4, 7

- HTML, 4
- HTML5, 4, 38

- ideographic (テキストのベースライン), 22
- image/png, 9
- ImageData オブジェクト, 31
 - の取得, 32
 - の生成, 33
 - の描画, 32
- Image オブジェクト, 30
- img 要素, 10
- input 要素, 37, 38

- JavaScript, 4

- keyCode プロパティ, 37

- left (テキストの配置), 21
- lineCap プロパティ, 17
- lineJoin プロパティ, 17, 18
- lineTo メソッド, 11
- lineWidth プロパティ, 6, 11, 17, 20

- max 属性, 38
- measureText メソッド, 23
- middle (テキストのベースライン), 22
- MIME タイプ, 9
- min 属性, 38
- miter (接続点の形状), 18
- miterLimit プロパティ, 17, 19
- moveTo メソッド, 11, 16

- no-repeat (パターンの繰り返し), 34

- offsetX プロパティ, 36
- offsetY プロパティ, 36
- onclick 属性, 36
- oninput 属性, 37
- onkeydown プロパティ, 37

- putImageData メソッド, 31, 32

- quadraticCurveTo メソッド, 14

- range (input 要素のタイプ), 38
- repeat (パターンの繰り返し), 34
- repeat-x (パターンの繰り返し), 34
- repeat-y (パターンの繰り返し), 34
- restore メソッド, 24

- RGBA, 8
- right (テキストの配置), 21
- rotate メソッド, 25
- round (接続点の形状), 18
- round (端点の形状), 17

- save メソッド, 24
- scale メソッド, 25, 26
- setInterval メソッド, 34
- setTransform メソッド, 25, 27
- shadowBlur プロパティ, 29
- shadowColor プロパティ, 29
- shadowOffsetX プロパティ, 29
- shadowOffsetY プロパティ, 29
- square (端点の形状), 17
- src 属性, 10
- start (テキストの配置), 21
- stroke メソッド, 11, 17
- strokeRect メソッド, 5, 6, 17, 18
- strokeStyle プロパティ, 7, 8, 11, 21, 28, 34
- strokeText メソッド, 20

- target プロパティ, 37
- text (input 要素のタイプ), 37
- textAlign プロパティ, 21
- textBaseline プロパティ, 22
- toDataURL メソッド, 9
- top (テキストのベースライン), 22
- transform メソッド, 25, 27
- translate メソッド, 25
- type 属性, 37, 38

- URL
 - 画像データの——, 9
- value 属性, 38

- width プロパティ, 23, 32
- width 属性, 4, 7

- x 軸, 5
- y 軸, 5

- アニメーション, 34
 - 移動の——, 35
- アルファ値, 8, 9
- アンダーライン, 23

- 位置
 - 影の——, 29
- 移動
 - のアニメーション, 35
 - 座標系の——, 25
- イベントオブジェクト, 36
- イベント属性, 36
- 色
 - 影の——, 29
 - 線の——, 8, 11
 - テキストの——, 21
 - 塗りつぶしの——, 7, 12

- 円弧, 15

- 扇形, 16
- 大きさ
 - Canvas の——, 4, 7

- 回転
 - 座標系の——, 26
- 拡大
 - 座標系の——, 26
- 角度
 - の単位, 16
- 影, 27, 29
 - の位置, 29
 - の色, 29
 - のぼかし度, 29
- 画像
 - の描画, 30
- 画像データ
 - の URL, 9
- カラーストップ, 28
- カレントクリッピング領域, 13
- カレントパス, 10
 - の塗りつぶし, 12
 - のリセット, 11
 - 線による——の描画, 11
- カレント描画状態, 24
- カレントポイント, 11

- キーコード, 37
- キーボード, 37

- グラディエント, 7, 27, 28
- グラフィックス
 - の消去, 6, 35
 - の装飾, 27
- クリッピング, 13
- クリッピング領域, 13

- 形状
 - 接続点の——, 18
 - 線の——, 17
 - 端点の——, 17

- コンテキスト ID, 5

- 座標系

- の移動, 25
- の回転, 26
- の拡大, 26
- の剪断, 27
- の変換, 25
- Canvas の—, 5
- サブパス, 10
 - の生成, 11
 - 閉じた—, 12
 - 開いた—, 12
- 三次ベジェ曲線, 14
- 取得
 - ImageData オブジェクトの—, 32
- 消去
 - グラフィックスの—, 6, 35
- 書字方向, 22
- 数値
 - を入力するフォーム要素, 38
- スタイル
 - のプロパティ, 7
 - 線の—, 7
 - 塗りつぶしの—, 7
- 制御点, 14
- 生成
 - ImageData オブジェクトの—, 33
 - サブパスの—, 11
- 接続点, 18
 - の形状, 18
- 線
 - によるカレントパスの描画, 11
 - による長方形の描画, 6
 - の色, 8, 11
 - の形状, 17
 - のスタイル, 7
 - の太さ, 6, 11, 20
- 線形グラディエント, 28
- センタリング, 21
- 剪断
 - 座標系の—, 27
- 装飾
 - グラフィックスの—, 27
- 単位
 - 角度の—, 16
- 端点, 17
 - の形状, 17
- 長方形, 5
 - の塗りつぶし, 5
 - のメソッド, 5
 - 線による—の描画, 6
- 直線, 11
- テキスト, 20
 - の色, 21
 - の配置, 21
 - のベースライン, 22
 - の横幅, 23
 - の輪郭を描画するメソッド, 20
 - を入力するフォーム要素, 37
 - を描画するメソッド, 20
- 度, 16
- 閉じた
 - サブパス, 12
- 二次ベジェ曲線, 14
- 塗りつぶし
 - の色, 7, 12
 - のスタイル, 7
 - カレントパスの—, 12
 - 長方形の—, 5
- 背景色, 4
- 配置, 21
 - テキストの—, 21
- パス, 10
- パターン, 7, 27, 34
- バットキャップ, 17
- ピクセル, 31
- ピクセル操作, 31
- 左寄せ, 21
- ビットマップ, 31
- 描画
 - ImageData オブジェクトの—, 32
 - 画像の—, 30
 - 線によるカレントパスの—, 11
 - 線による長方形の—, 6
- 描画コンテキスト, 5, 24
- 描画状態, 24
- 描画状態スタック, 24
- 開いた
 - サブパス, 12
- フォーム要素, 37
 - 数値を入力する—, 38
 - テキストを入力する—, 37
- 太さ
 - 線の—, 6, 11, 20
- プロジェクティングスクエアキャップ, 17
- プロパティ
 - スタイルの—, 7
- ベースライン, 22
 - テキストの—, 22
- ベジェ曲線, 14
- ベベルジョイン, 18

変換

- 座標系の——, 25

- 変換行列, 25

- 放射状グラディエント, 28, 29

- ボーダー, 4

- ぼかし度

- 影の——, 29

- 補色, 32

- マイタージョイン, 18

- マイター長, 19

- マイターリミット率, 19

- 右寄せ, 21

メソッド

- 長方形の——, 5

- テキストの輪郭を描画する——, 20

- テキストを描画する——, 20

横幅

- テキストの——, 23

- ラウンドキャップ, 17

- ラウンドジョイン, 18

- ラジアン, 16

リセット

- カレントパスの——, 11

輪郭

- テキストの——を描画するメソッド, 20