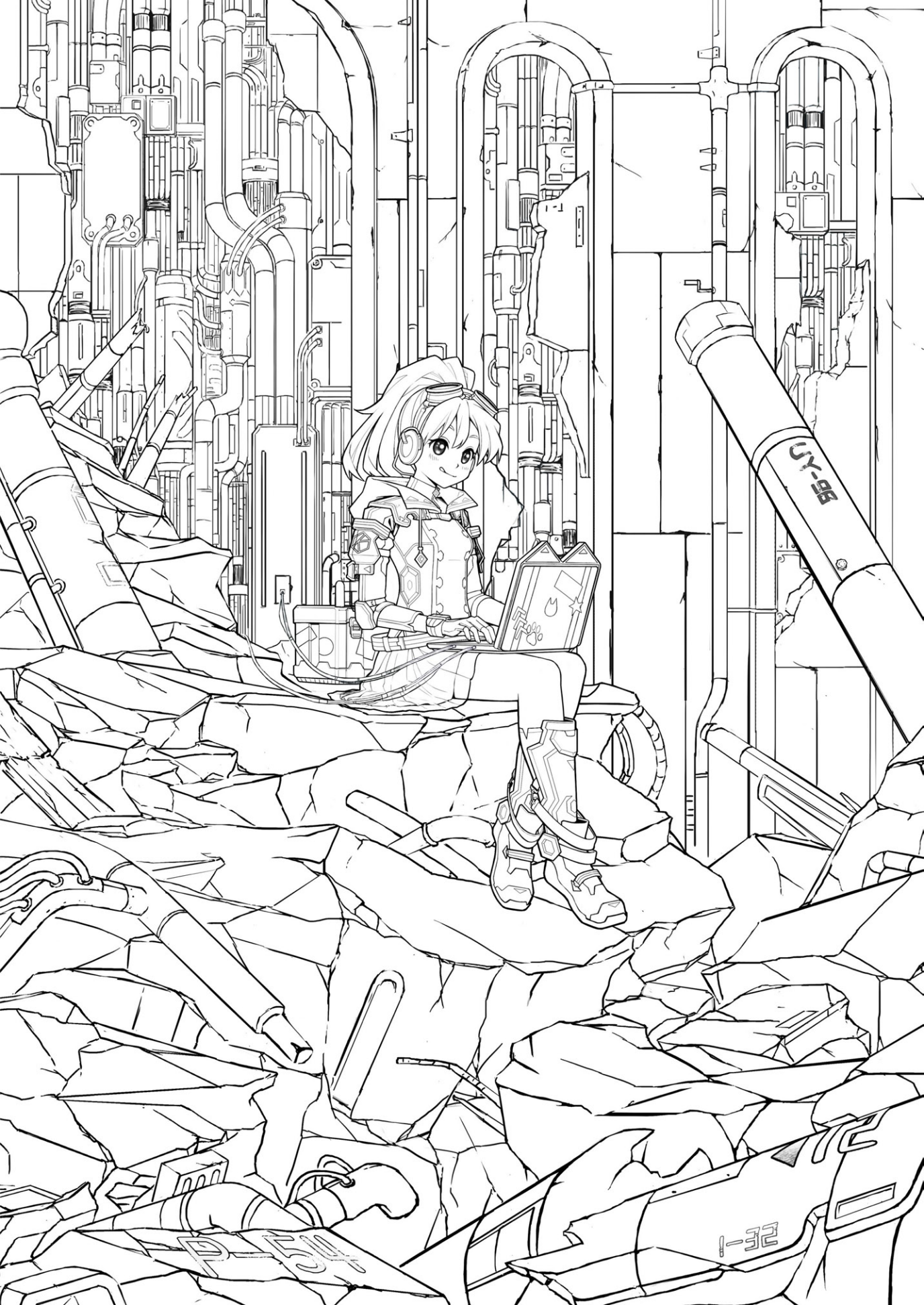
KLab Tech Book v6

2020 SPRING

Unity Ray Marching Unity 2D Animation
Arduino Uno Python Rust Lambda@Edge
Competitive Programming Barcode
Emacs Lisp Automation Vulkan
Built-in Function Code128 Path Tracing
IK Animation React Programming Contest


UY-88

KLab Tech Book Vol. 6

2020-03-07 版　　**KLab** 技術書サークル 発行

はじめに

このたびは本書をお手に取っていただきありがとうございます。本書は KLab 株式会社の有志にて作成された KLab Tech Book の第 6 弾です。

KLab 株式会社では主にスマートフォン向けのゲームを開発していますが、本書ではこれまでどおり社内で執筆者を募り、著者の興味や得意分野をベースに各自好きな技術について好きなように書き執筆者同士（＋α ）レビューを行なった記事を収録しています。業務に少しだけ関係のあることもあり、完全に趣味の内容もあります。

第 6 弾ともなると継続して執筆に参加してきた著者もいます。特定の技術領域について継続して執筆している著者がいたり、回ごとに執筆する内容の技術領域を変えている著者もいたり。また、今回はじめて技術同人誌を書いてみたという著者もいます。気になった記事の著者が過去執筆していたか？ 過去執筆していた記事でどんな内容を書いていたのか？ 巻末の QR コードから追いかけてみるのも楽しいのではないかと思います。

また、既にご覧頂いている表紙ですが、これもこれまでに続き、社内のイラストレーター、デザイナーが世界観やキャラクターをイチから制作してくれたものです。物理版では今回初となる箔押しにも挑戦しております。物理版をお持ちの方はぜひそのデザインをよく見てみて頂けると嬉しいです。表紙も本文も一冊まるごと、読者のみなさまに楽しんで頂ければさいわいです。

水澤 絹子

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

NVIDIA OptiX で『レイマーチング×パストレーシング』による物理ベースレンダラーを自作する

Sho HOSODA / @gam0022

▲図 1.1　Redflash によるレンダリング結果

本章では、筆者自作の物理ベースレンダラー **Redflash**[*1] を題材にして、GPU と RT コアを用いた高速なレイトレーシングについて紹介します。

[*1] https://github.com/gam0022/redflash

Redflash は **NVIDIA OptiX 6.0** 上で実装したパストレーシングによる物理ベースの GPU レンダラーです。Redflash では、ポリゴンとレイマーチングが混在したシーンを一貫して描画できます。図 1.1 の Redflash による描画結果では、画面中央の天使像がポリゴンで表現されたオブジェクトであり、背景の幾何学的なフラクタルが距離関数で表現されたレイマーチングによるオブジェクトです。ポリゴンやレイマーチングといったプリミティブを区別せずに、まったく同じアルゴリズムでライティングやシェーディングを計算するように設計しました。

また、本章は技術書典 3『KLab Tech Book Vol. 1』[*2]の第 1 章『物理ベースレンダラーを Rust 実装して、表紙絵をレンダリングした話』と技術書典 5『KLab Tech Book Vol. 3』[*3]の第 7 章『物理ベースレンダラーを Rust 実装して、ちょっと高速化した話』の続編です。

1.1 用語のおさらい

基本的な用語についておさらいします。おさらいが長くなってしまうため、細かい用語までは触れませんので、もっと詳しく知りたい方は前回までの章をご参照ください。

強力な交差判定の手法『レイマーチング』

レイマーチングは、**距離関数（3D 空間上の座標を入力すると、物体への最短距離を出力する関数）で定義されたシーンに対し、レイの交差を判定する手法**です。図 1.1 の背景は **Mandelbox** という有名なフラクタル図形です。複雑なディテールを持ったフラクタル図形も、距離関数によって比較的短いソースコードで定義ができます。レイマーチングを用いると、距離関数によって定義した形状をそのまま交差判定ができるため、幾何学的な形状が欲しいときには超強力な手法といえます。

写実的に描画する手法『パストレーシング』

パストレーシングを簡単にいうと、**現実世界の光の振る舞いを簡略化してシミュレートすることで、写実的なレンダリングができる 3D の描画手法**です。

3DCG の描画方法は**レイトレーシング法**と**ラスタライズ法**の大きく 2 つに分類できます。レイトレーシング法ではカメラからレイを飛ばし、レイがシーン中の物体に衝突したらシーンを描画し、衝突しなかったら背景を描画します。そして物体表面がどのくらい照らされているかを計算し、色付けをしていきます。この色付け処理を**シェーディング**と呼びます。

パストレーシングでは、このシェーディングの処理として光の伝達・挙動を記述した**レ**

[*2] http://klabgames.tech.blog.jp.klab.com/techbook/KLab-Tech-Book-Vol-1-1.2-ebook.pdf

[*3] http://klabgames.tech.blog.jp.klab.com/techbook/KLabTechBook_Vol3.pdf

ンダリング方程式を用います。レンダリング方程式を正確に計算することで大域照明を考慮した写実的な描画ができますが、実際のシーンのレンダリングにおいて解析的に正確な計算をすることは非現実的です。そこで、**モンテカルロ積分**（乱数を使ったシミュレーションを繰り返して数値的に積分を計算する手法）を用いてレンダリング方程式を解きます。以上をまとめると、**パストレーシングはレイトレーシングに分類される手法**で、**モンテカルロ積分を用いてレンダリング方程式を計算することでシェーディングを行います**。

1.2 NVIDIA が提供するレイトレーシング用 API『OptiX』

OptiX[*4]は、CUDA 基盤上で動作する、NVIDIA が提供するレイトレーシング用の API です。CUDA は、GPU 向けの汎用並列コンピューティングプラットフォームであり、OptiX と同様に NVIDIA が提供しています。レイトレーシングの衝突判定やシェーディングといった各パイプラインの処理を CUDA C という言語で記述することで GPU 上でレイトレーシングができます。

CUDA C は C++ 言語に近い文法で記述できるコンピュートシェーダーのようなもので、拡張子は.cuです。C++ と同じような文法ですが、GPU 上で動作するため、標準出力ができない・再帰関数が使えない等のコンピュートシェーダーと同様の制約があります。

OptiX 用語で、レイトレーシングの各パイプラインの処理を定義する CUDA C のことを **Program** と呼びます。代表的な Program を紹介します。

Ray Generation

レイトレーシングのパイプラインのエントリーポイントです。ピクセルに対応するレイを生成することで、ピンホールカメラ等のカメラを定義できます。

Closest Hit

レイが最も近くにある衝突点にヒットしたときに呼ばれます。ここでマテリアルの定義に応じたシェーディングを実装します。

Any Hit

レイがオブジェクトに交差するたびに何度も呼ばれます。アルファテストや影の計算に利用できる他、パストレーシングの **NEE**（Next Event Estimation）のためのシャドウレイの実装にも使われます。

Bounding Box

BVH の空間構造を構築するときに呼ばれます。プリミティブのワールド空間での Bounding Box を定義します。

Intersection

BVH の探索（トラバーサル）中に呼ばれます。レイとレイプリミティブの交差判定を実装します。

[*4] NVIDIA OptiX Ray Tracing Engine `https://developer.nvidia.com/optix`

Miss

レイがすべてのオブジェクトに交差しなかったときに呼ばれます。Skybox の定義に利用できます。

レイトレーシングの処理のフローは OptiX によって定義されているため、OptiX の作法に従う必要がありますが、各 Program を自分で定義することで、衝突判定やシェーディングを自由にカスタマイズができます。

1.3 BVH の探索を加速する『RT コア』

『KLab Tech Book Vol. 3』[*5]では **BVH**（Bounding Volume Hierarchy）による衝突判定の高速化を紹介しました。BVH はレイトレーシングに最適な空間分割の手法で、シーンに大量のオブジェクトやポリゴンが含まれるほど効果を発揮します。シーンに含まれるポリゴンの数を n としたとき、BVH なしの総当たりで衝突判定する場合、計算量のオーダーは $O(n)$ となります。一方、BVH を用いる場合は $O(log\ n)$ になります。

効率のよい BVH を構築するのもなかなか奥が深い問題で、あらゆるシーンにおいて BVH の構築と探索のパフォーマンスを両立させるためには高度な知識と技術が必要です。

OptiX の API には BVH の構築と探索が含まれているので、頭を悩ませる BVH については考えずに済みます。裏を返すと、OptiX を使うなら BVH の構築と探索は手を出せない領域になってしまっているともいえます。

とはいえ、あらゆる部分をプログラマブルにしてしまうと、ハードウェア支援が受けづらくなり、パフォーマンスが犠牲になってしまいます。BVH の構築と探索はレイトレーシングの実装においても、マテリアルやプリミティブの定義と比較して、カスタマイズしたいという需要が少ない部分なので、ここを固定にすることでパフォーマンス向上する戦略は理にかなっていると筆者は思っていました。

さらに、**RT コア**の登場によって、NVIDIA のレイトレーシングのハードウェア支援の戦略が明らかになり、この考えはより強化されました。NVIDIA が 2018 年に発表した Turing アーキテクチャの GPU では、RT コアと呼ばれるレイトレーシング専用ユニットが搭載されるようになりました。RT コアの正体は BVH の探索を行うための専用ハードウェアです。SIGGRAPH 2018 の NVIDIA の基調講演で発せられた「10 Giga Rays/s」という宣伝文句が話題になりましたが、RT コアのハードウェア支援によって、短時間に大量のレイを高速に飛ばせるようになったということですね。

なんと OptiX 6.0 以降のバージョンでは、RT コアもサポートされているため、プログラマーが特に意識をせずに RT コアの恩恵を受けることができます。素晴らしいですね！

[*5] http://klabgames.tech.blog.jp.klab.com/techbook/KLabTechBook_Vol3.pdf

1.4 Redflash の実装解説

ここからは筆者自作の物理ベースレンダラー Redflash を題材にして、実装の解説を行います。

OptiX 上でレイマーチングを実装する

OptiX にはユーザ定義のプリミティブを定義する機能が備わっているため、特殊なハックを使わずともレイマーチングを実装できます。具体的には次の 2 つの Program を定義する必要があります。

- Bounding Box
- Intersection

ここから実装の概要について簡単に紹介します。実装の詳細については、ソースコード[*6]を参照してください。

リスト 1.1 は Bounding Box を定義する Program です。CPU で計算した **AABB**（Axis-Aligned Bounding Box）の min と max をそのまま返すだけの内容が無いコードです。プリミティブで共通した値を返せばよい箇所は、なるべく CPU で計算して、GPU の負荷を下げる狙いがあります。

▼リスト 1.1　レイマーチングの Bounding Box Program

```
rtDeclareVariable(float3, aabb_min, , );
rtDeclareVariable(float3, aabb_max, , );

RT_PROGRAM void bounds(int, float result[6])
{
    optix::Aabb* aabb = (optix::Aabb*)result;
    aabb->m_min = aabb_min;
    aabb->m_max = aabb_max;
}
```

リスト 1.2 は Intersection を定義する Program です。すこしボリュームがありますが、内容は非常にシンプルです。最重要は `intersect()`関数で、ここでレイマーチングをしています。`map()`は最終的な距離関数です。そして `dMandelFast()`関数は Mandelbox と呼ばれるフラクタル図形の距離関数で、`map()`から呼び出される距離関数の具体的な実装です。

ちなみに CUDA C では Swizzle Operation が使えませんでした。`get_xyz()`関数と `set_xyz()`関数は、それぞれ 4 次元ベクトルに対して `.xyz`の読み込みと書き込みの Swizzle Operation を再現するためのヘルパー関数です。

[*6] https://github.com/gam0022/redflash/blob/master/redflash/intersect_raymarching.cu

▼リスト 1.2　レイマーチングの Intersection Program

```
rtDeclareVariable(float, scene_epsilon, , );
rtDeclareVariable(float3, geometric_normal, attribute geometric_normal, );
rtDeclareVariable(float3, shading_normal, attribute shading_normal, );
rtDeclareVariable(optix::Ray, ray, rtCurrentRay, );

rtDeclareVariable(float3, center, , );
rtDeclareVariable(float3, local_scale, , );
rtDeclareVariable(float3, texcoord, attribute texcoord, );

RT_FUNCTION float3 get_xyz(const float4 &a)
{
    return make_float3(a.x, a.y, a.z);
}

RT_FUNCTION void set_xyz(float4 &a, const float3 &b)
{
    a.x = b.x;
    a.y = b.y;
    a.z = b.z;
}

RT_FUNCTION float dMandelFast(float3 p, float scale, int n)
{
    float4 q0 = make_float4(p, 1.);
    float4 q = q0;

    for (int i = 0; i < n; i++)
    {
        // q.xyz = clamp(q.xyz, -1.0, 1.0) * 2.0 - q.xyz;
        float3 q_xyz = get_xyz(q);
        set_xyz(q, clamp(q_xyz, -1.0, 1.0) * 2.0 - q_xyz);

        // q = q * scale / clamp( dot( q.xyz, q.xyz ), 0.3, 1.0 ) + q0;
        q_xyz = get_xyz(q);
        q = q * scale / clamp(dot(q_xyz, q_xyz), 0.3, 1.0) + q0;
    }

    // return length( q.xyz ) / abs( q.w );
    return length(get_xyz(q)) / abs(q.w);
}

float map(float3 p)
{
    return dMandelFast((p - center) / local_scale, 2.76, 20) *
        min(min(local_scale.x, local_scale.y), local_scale.z);
}

#define calcNormal(p, dFunc, eps) normalize(\
  make_float3( eps, -eps, -eps) * dFunc(p + make_float3( eps, -eps, -eps)) + \
  make_float3(-eps, -eps,  eps) * dFunc(p + make_float3(-eps, -eps,  eps)) + \
  make_float3(-eps,  eps, -eps) * dFunc(p + make_float3(-eps,  eps, -eps)) + \
  make_float3( eps,  eps,  eps) * dFunc(p + make_float3( eps,  eps,  eps)))

RT_PROGRAM void intersect(int primIdx)
{
    float eps;
    float t = ray.tmin, d = 0.0;
    float3 p = ray.origin;

    for (int i = 0; i < 300; i++)
    {
        p = ray.origin + t * ray.direction;
        d = map(p);
        t += d;
        eps = scene_epsilon * t;
        if (abs(d) < eps || t > ray.tmax)
        {
            break;
        }
    }
```

```
    if (t < ray.tmax && rtPotentialIntersection(t))
    {
        shading_normal = geometric_normal = calcNormal(p, map, scene_epsilon);
        texcoord = make_float3(p.x, p.y, 0);
        rtReportIntersection(0);
    }
}
```

OptiX 上でパストレーシングを実装する

OptiX でマテリアルを定義するためには、Closest Hit Program を実装する必要があります。Closest Hit Program の中で `rtTrace()`を呼び出すことで再帰的にレイを追跡できますが、パフォーマンスの観点では、再帰呼び出しよりも、ループの方が効率的です。Redflash では、パフォーマンス向上のためにプライマリレイを生成する Ray Generation Program でレイのバウンスを処理するためのループを定義し、ループの中で `rtTrace()` を呼び出す方針で実装しました。

ここから実装の概要について簡単に紹介します。パストレーシング全体の実装はかなり長くなってしまいますので、重要なコードのみを抜粋して紹介します。実装の詳細については、ソースコード[*7]を参照してください。

リスト 1.3 はパストレーシングの Ray Generation Program です。処理内容としては次のことを行っています。

- スクリーンのピクセルに対応するプライマリレイの生成
- ループを用いて再帰的にレイを追跡
- 最終的な結果をバッファに書き込み

`sample_per_launch`のループはパストレーシングのサンプル回数に対応します。その中にある無限ループで再帰的にループをしていて、光源にヒットするか、反射回数の上限になるまでレイを追跡しています。

▼リスト 1.3　パストレーシングの Ray Generation Program

```
RT_PROGRAM void pathtrace_camera()
{
    size_t2 screen = output_buffer.size();
    float3 result = make_float3(0.0f);
    float3 albedo = make_float3(0.0f);
    float3 normal = make_float3(0.0f);
    unsigned int seed =
        tea<16>(screen.x * launch_index.y + launch_index.x, total_sample);

    for (int i = 0; i < sample_per_launch; i++)
    {
        float2 subpixel_jitter =
            make_float2(rnd(seed) - 0.5f, rnd(seed) - 0.5f);
        float2 d = (make_float2(launch_index) + subpixel_jitter) /
```

[*7] https://github.com/gam0022/redflash/blob/master/redflash/redflash.cu

```
            make_float2(screen) * 2.f - 1.f;
        float3 ray_origin = eye;
        float3 ray_direction = normalize(d.x*U + d.y*V + W);

        // Initialze per-ray data
        PerRayData_pathtrace prd;
        prd.radiance = make_float3(0.0f);
        prd.attenuation = make_float3(1.0f);
        prd.done = false;
        prd.seed = seed;
        prd.depth = 0;

        // Each iteration is a segment of the ray path.  The closest hit will
        // return new segments to be traced here.
        for (;;)
        {
            Ray ray = make_Ray(ray_origin, ray_direction, RADIANCE_RAY_TYPE,
                scene_epsilon, RT_DEFAULT_MAX);
            prd.wo = -ray.direction;
            rtTrace(top_object, ray, prd);

            if (prd.done || prd.depth >= max_depth)
            {
                break;
            }

            if (prd.depth == 0)
            {
                albedo += prd.albedo;
                normal += prd.normal;
            }

            // Update ray data for the next path segment
            ray_origin = prd.origin;
            ray_direction = prd.direction;

            prd.depth++;
        }

        result += prd.radiance;
    }

    //
    // Update the output buffer
    //
    float3 normal_eyespace = (length(normal) > 0.0f) ?
        normalize(normal_matrix * normal) : make_float3(0.0, 0.0, 1.0);

    float inv_sample_per_launch = 1.0f / static_cast<float>(sample_per_launch);
    float3 pixel_liner = result * inv_sample_per_launch;
    float3 pixel_albedo = albedo * inv_sample_per_launch;
    float3 pixel_normal = normal_eyespace;

    if (frame_number > 1)
    {
        float a = static_cast<float>(sample_per_launch) /
            static_cast<float>(total_sample + sample_per_launch);
        pixel_liner =
            lerp(make_float3(liner_buffer[launch_index]), pixel_liner, a);
    }

    float3 pixel_output = use_post_tonemap ? pixel_liner :
        linear_to_sRGB(tonemap_acesFilm(pixel_liner * tonemap_exposure));

    // Save to buffer
    liner_buffer[launch_index] = make_float4(pixel_liner, 1.0);
    output_buffer[launch_index] = make_float4(pixel_output, 1.0);

    // NOTE: デノイズ用の情報は 1 フレーム目しか更新しない
    // NOTE: DOF とかモーションブラーなら毎フレーム更新した方がいいのかもしれない
    if (frame_number == 1)
    {
```

```
        input_albedo_buffer[launch_index] = make_float4(pixel_albedo, 1.0f);
        input_normal_buffer[launch_index] = make_float4(pixel_normal, 1.0f);
    }
}
```

リスト 1.4 はパストレーシングの Closest Hit Program です。Closest Hit Program はレイが最も近くにある衝突点にヒットしたときに呼ばれ、ヒットしたオブジェクトのマテリアルの **BRDF**（双方向反射率分布関数）に応じて確率的に次のレイの方向を決定したり、光源に衝突した場合は光を蓄積します。

パストレーシングの実装では、マテリアルごとに異なる処理は、BRDF に応じたサンプリングや **PDF**（確率密度関数）だけで、それ以外の処理は共通化できます。そこで、マテリアルごとに異なる処理を **Bindless Callable Program** と呼ばれる関数ポインタのような機能を用いて切り替える設計としました。これにより、すべてのマテリアルで共通の Closest Hit Program を使うことができるため、共通化によって実装がきれいになるだけでなく、divergence を軽減して GPU の実行効率の向上も期待できます。

▼リスト 1.4　パストレーシングの Closest Hit Program

```
RT_PROGRAM void closest_hit()
{
    float3 world_shading_normal = normalize(
        rtTransformNormal(RT_OBJECT_TO_WORLD, shading_normal));
    float3 world_geometric_normal = normalize(
        rtTransformNormal(RT_OBJECT_TO_WORLD, geometric_normal));
    float3 ffnormal = faceforward(world_shading_normal,
        -ray.direction, world_geometric_normal);

    float3 hitpoint = ray.origin + t_hit * ray.direction +
        ffnormal * scene_epsilon * 10.0;

    State state;
    state.hitpoint = hitpoint;
    state.normal = world_shading_normal;
    state.ffnormal = ffnormal;

    MaterialParameter mat = sysMaterialParameters[material_id];

    current_prd.radiance += mat.emission * current_prd.attenuation;
    current_prd.wo = -ray.direction;
    current_prd.albedo = mat.albedo;
    current_prd.normal = ffnormal;

    current_prd.origin = hitpoint;
    current_prd.specularBounce = false;

    // Direct light Sampling
    if (!current_prd.specularBounce && current_prd.depth < max_depth)
    {
        current_prd.radiance += DirectLight(mat, state);
    }

    // BRDF Sampling
    prgs_BSDF_Sample[bsdf_id](mat, state, current_prd);
    prgs_BSDF_Pdf[bsdf_id](mat, state, current_prd);
    float3 f = prgs_BSDF_Eval[bsdf_id](mat, state, current_prd);

    if (current_prd.pdf > 0.0f)
    {
        current_prd.attenuation *= f / current_prd.pdf;
    }
```

```
    else
    {
        current_prd.done = true;
    }
}
```

実際に BRDF のサンプリングや PDF の評価を行う実装については `bsdf_diffuse.cu`[*8]と `bsdf_disney.cu`[*9]に実装しました。それぞれ完全拡散の BRDF と Disney Principled BRDF です。

1.5 まとめ

本章では、OptiX を用いたレイマーチングとパストレーシングの実装について紹介しました。OptiX では、レイトレーシングの各パイプラインを CUDA C 言語で記述することで、独自のプリミティブやマテリアルを定義できるため、レイマーチングやパストレーシングといった複雑なレイトレーシングも実装が可能です。OptiX の API に乗っかることで、0 から実装するより少ない労力で GPU と RT コアを用いた高速なレイトレーシングを実現できます。

1.6 おわりに

ちなみに Redflash は**レイトレ合宿**というイベントに参加するために開発しました。

レイトレ合宿は完全自作のレイトレーサーを走らせて画像の美しさや技術力を投票で競うイベントです。参加条件としてレンダラーを自作する必要があるというだけで面白い（というより、エクストリームな）イベントなのですが、レンダリングの制限時間が毎年どんどん短縮されているのも注目ポイントです。第 1 回のレンダリング合宿では制限時間が 1 時間だったのですが、第 7 回となる今年は 60 秒制限でした。この制限時間はレンダラーを起動してから画像を保存するまでの時間なので、シーンの読み込みからレンダリングをすべて含めて 60 秒で完了させなくてはなりません。最初に紹介した図 1.1 もわずか 60 秒で描画したレンダリング結果です。

レイトレ合宿の参加報告[*10]とアドベントカレンダーの記事[*11]を筆者のブログに投稿していますので、興味がありましたらご覧ください。レイトレ合宿の参加報告では、衝突判定の高速化や **Deep Learning Denoising**（ディープラーニングを用いたパストレーシングの高周波ノイズの除去）など Redflash の高速化の工夫について紹介しています。

[*8] https://github.com/gam0022/redflash/blob/master/redflash/bsdf_diffuse.cu

[*9] https://github.com/gam0022/redflash/blob/master/redflash/bsdf_disney.cu

[*10] レイトレ合宿 7 でレイマーチング対応の GPU パストレーサーを実装しました！ https://gam0022.net/blog/2019/09/18/rtcamp7/

[*11] NVIDIA ® OptiX 上で『レイマーチング×パストレーシング』による物理ベースレンダラーを実装した https://gam0022.net/blog/2019/08/05/optix-raymarching-pathtracing/

1.7 OptiX の参考資料の紹介

OptiX を学習するにあたって、参考になった資料を紹介します。

optix - uimac 実装メモ

http://memo.render.jp/optix

shocker-0x15/VLR Wiki

https://github.com/shocker-0x15/VLR/wiki

OptiX QuickStart - NVIDIA Developer Documentation

https://docs.nvidia.com/gameworks/content/gameworkslibrary/optix/optix_quickstart.htm

第 2 章

Unity の描画ラインをハイジャックして遊ぶ

Shinya Naganuma / @Pctg_x8

Unity のネイティブプラグインで描画の一部をジャックして、色々遊んでみます[*1]。

環境は Windows 10 + Unity 2019.2 + Vulkan バックエンド で、言語は C++ ではなく Rust を使用します。

2.1 Unity のネイティブプラグインについて

Unity にはプラットフォーム固有のコードや、C++ などで作成したマシンネイティブなコードをユーザーが作成して、組み込むことのできる機能が存在します。それが「ネイティブプラグイン」と呼ばれているもので、独自のサウンドエフェクトをより処理効率のよい形で実装したり、OpenCV などのネイティブなライブラリを利用したりすることができます。

このネイティブプラグインはおよそありとあらゆるプログラムを書くことが可能です。Unity の描画システムに割り込んで、描画バッファを横取りしたり任意の図形をオーバードローしたりもできます。今回はそのポテンシャルを最大限引き出してみたいと思います。

2.2 ネイティブ（レンダリング）プラグインを書く

今回利用するネイティブプラグインは、レンダリングに関係するプラグインになりますので「Low-level Native Plug-in Interface[*2]」に沿った実装が必要になります。このインターフェイスでは、初期化を `UnityPluginLoad`、後始末を `UnityPluginUnload`と命名

[*1] 遊んだ跡はこちら: `https://github.com/Pctg-x8/native-render-intercept-test`

[*2] `https://docs.unity3d.com/Manual/NativePluginInterface.html`

された関数で行うことになっていますので、この 2 つを最低限エクスポートする必要があります。これらはリスト 2.1 に示すようなインターフェイスになっています。

▼リスト 2.1　UnityPluginLoad/UnityPluginUnload

```
pub extern "system" fn UnityPluginLoad(ifs: *mut unity::IUnityInterfaces);
pub extern "system" fn UnityPluginUnload();
```

UnityPluginLoadでは、各種プラグインの初期化に必要な IUnityInterfacesへのポインタが渡されます。一方で、UnityPluginUnloadでは渡されません。そのため、IUnityInterfacesおよびそこから派生して取得できるオブジェクトはプラグイン側でグローバルに保持しておく必要があります。

Unity のレンダリングシステムに割り込むためには、IUnityInterfaces::get_interfaceを使用して IUnityGraphicsを取得する必要があります。IUnityInterfaces::get_interfaceの定義をリスト 2.2 に示します。

▼リスト 2.2　IUnityInterfaces::get_interface

```
#[repr(C)]
pub struct IUnityInterfaces
{
    ...
    pub get_interface: extern "system" fn(guid: UnityInterfaceGUID)
        -> *mut IUnityInterface,
    ...
}
```

IUnityInterfaces::get_interfaceで取得できるオブジェクトの型は、**COM**（Component Object Model）のように GUID を用いて識別されています。

Unity のレンダリングプラグインはイベントドリブンな構成を取っています。Unity から送られてくるグラフィックスデバイス関係のイベントに反応するには IUnityGraphics::register_device_event_callbackに関数を登録します。プラグインの Unload 時など、反応が必要なくなる場合は IUnityGraphics::unregister_device_event_callbackを呼び出して関数の登録を消します。イベントハンドラとなる関数はリスト 2.3 のようになっています。見た目のとおり、イベントタイプが引数で渡ってくるので、プラグイン側で適切な処理を行います。

▼リスト 2.3　イベントハンドラ

```
pub type IUnityGraphicsDeviceEventCallback =
    extern "system" fn(event_type: UnityGfxDeviceEventType);
```

ここまでをまとめると、リスト 2.4 に示すコードでプラグインを初期化できます。なお、IUnityGraphicsの GUID は IUnityGraphics::GUIDに定義されているものとして

います。UnityPluginUnloadで後始末をする際は、gfx_ifを TLS[*3]に格納してそれを使用します。

▼リスト 2.4　UnityPluginLoad

```
thread_local!{
    static INTERFACES: Cell<*mut IUnityInterfaces> = Cell::new(null_mut());
    static GFX_IF: Cell<*mut IUnityGraphics> = Cell::new(null_mut());
}

pub extern "system" fn UnityPluginLoad(ifs: *mut IUnityInterfaces)
{
    INTERFACES.with(|v| v.set(ifs));
    let gfx_if = unsafe
    {
        ((*ifs).get_interface)(IUnityGraphics::GUID)
            as *mut IUnityGraphics
    };
    GFX_IF.with(|v| v.set(gfx_if));
    unsafe { ((*gfx_if).register_device_event_callback)(gfx_event_handler); }

    // グラフィックスデバイスが初期化された後にプラグインのロードが走っても
    // 確実に初期化させるため、手動でも呼び出す
    // ref: https://docs.unity3d.com/Manual/NativePluginInterface.html
    gfx_event_handler(kUnityGfxDeviceEventInitialize);
}

pub extern "system" fn gfx_event_handler(event_type: UnityGfxDeviceEventType)
{
    ...
}
```

イベントハンドラの登録ができたので、次はその中の実装をしていきます。2019.2 の時点で定義されているイベントタイプは次の 4 つですが、下 2 つは Direct3D9 特有のイベントなので、Direct3D9 をグラフィックス API としてサポートしない場合は実装する必要はありません。

kUnityGfxDeviceEventInitialize
: グラフィックスデバイスの初期化タイミングで呼ばれる。IUnityGraphics::get_rendererでバックエンド API を判定して適切に初期化を行う。

kUnityGfxDeviceEventShutdown
: グラフィックスデバイスの後始末のタイミングで呼ばれる。

kUnityGfxDeviceEventBeforeReset
: 推測だが、何らかの理由により IDirect3DDevice9::Resetを発行する前に呼ばれる。IDirect3DDevice9::Resetはテクスチャなどすべてのリソースを無効化するため、その内容をメインメモリに書き戻して保存するなどの処理をここでやる。

kUnityGfxDeviceEventAfterReset
: IDirect3DDevice9::Resetの呼び出し後に呼ばれる。上記イベントで退避したテクスチャデータを元にしたリソースの復元などの処理をここで行う。

今回は対応するグラフィックス API を Vulkan に限定しているため、IUnityGraphic

[*3] Thread Local Storage。*mut は Send ではないため、static な RwLock で囲ってもつことができない

s::get_rendererで kUnityGfxRendererVulkan以外が返ってきた場合は無視しています。他プラットフォーム（PS4 など）でも使えるものを作る場合はもう少し描画オブジェクトの持ち方を工夫する必要があります。イベントハンドラの今回の例をリスト 2.5 に示します。

▼リスト 2.5　イベントハンドラ

```
lazy_static!{
    static ref GRAPHICS_DEVICE: RwLock<Option<VkRenderingInterceptor>> =
        RwLock::new(None);
}
pub extern "system" fn gfx_event_handler(event_type: UnityGfxDeviceEventType)
{
    if event_type == kUnityGfxDeviceEventInitialize
    {
        let renderer_type = GFX_IF
            .with(|o| unsafe { ((*o.get()).get_renderer)() });
        if renderer_type != kUnityGfxRendererVulkan
        {
            // Renderer Type is not supported!
            return;
        }

        *GRAPHICS_DEVICE.write().unwrap() = Some(VkRenderingInterceptor::new());
    }
    else if event_type == kUnityGfxDeviceEventShutdown
    {
        *GRAPHICS_DEVICE.write().unwrap() = None;
    }
}
```

Unity 側で生成した Vulkan のインスタンスオブジェクトなどは IUnityGraphicsVulkanを経由して取得できます。IUnityGraphicsVulkanも IUnityInterfaces::get_interfaceを経由して取得します。

以上でプラグインの初期化/後始末は完了です。

実際に描画コマンド列に割り込んで処理をするには、UnityRenderingEvent型に沿った関数をコールバックとして GL.IssuePluginEventを C#側から呼んでもらうようにします。UnityRenderingEventの定義をリスト 2.6 に示します。event_idには GL.IssuePluginEventに渡したイベント ID が入ります。

▼リスト 2.6　UnityRenderingEvent

```
pub type UnityRenderingEvent = extern "system" fn(event_id: c_int);
```

GL.IssuePluginEventでは引数に UnityRenderingEvent型の関数ポインタを渡す必要があるため、そのポインタを返す関数もエクスポートする必要があります。合わせると、リスト 2.7 とリスト 2.8 のようになります。

▼リスト 2.7　プラグイン側コード

```
#[no_mangle]
pub extern "system" fn rendering_event_ptr() -> UnityRenderingEvent
{
    rendering_event
}

extern "system" fn rendering_event(event_id: c_int)
{
    ...
}
```

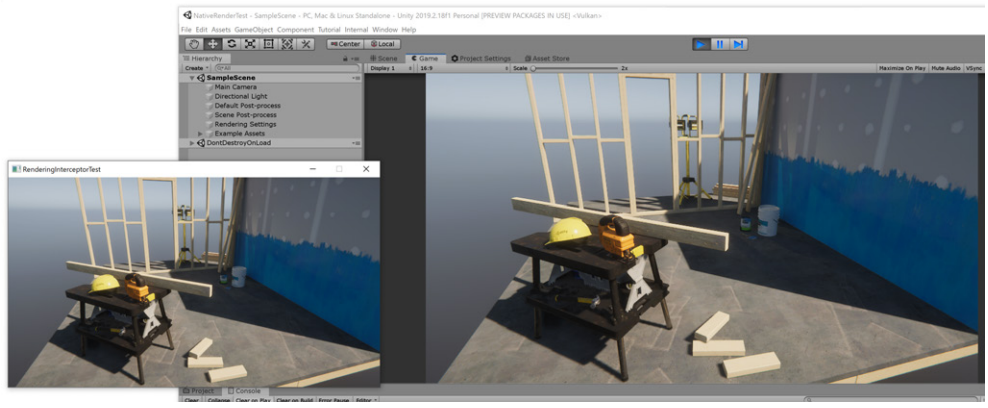
▼リスト 2.8　C#側コード

```
#if !UNITY_EDITOR
    [DllImport("RenderingInterceptor")]
    private static extern IntPtr rendering_event_ptr();
#endif
    ...
        GL.IssuePluginEvent(rendering_event_ptr(), 1);
```

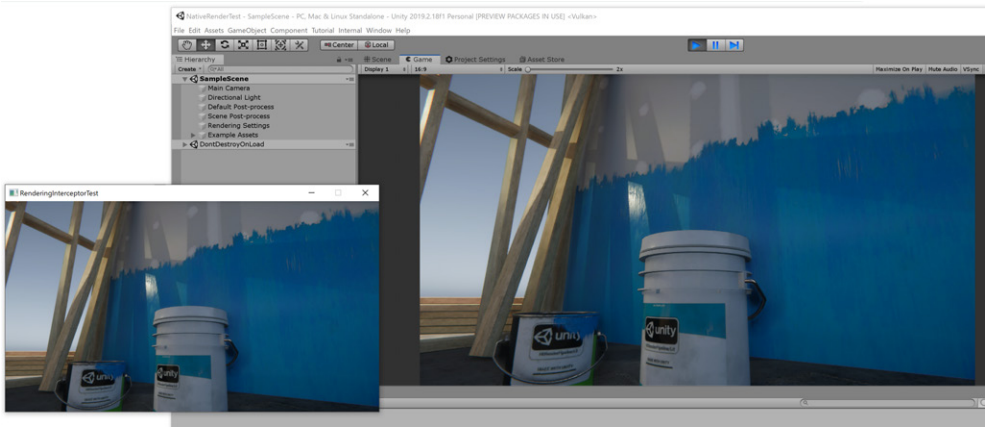
2.3　実践編 - Screen Casting

ネイティブプラグインについての説明は以上にして、ここからは実際にネイティブプラグインを書いて Unity の範囲を大きく超えた遊びをしていきます。

今回は、ネイティブプラグイン内でウィンドウを開き、そこに Unity のレンダリング内容をリアルタイムにキャストするだけの簡単なデモを作りました。



▲図 2.1　スクリーンショット



▲図 2.2　別視点から

流れとしては難しいことはしておらず、GL.IssuePluginEventのタイミングで Unity の現在の描画ターゲットに紐づいたテクスチャをプラグイン側の Swapchain のイメージオブジェクトに Blit しているだけです。Unity の現在の描画ターゲットは Graphics.activeColorBufferで取得できます。ただしこのコードで取得できるオブジェクト (RenderBuffer) はそのままではネイティブプラグインに渡すことができず、RenderBuffer.GetNativeRenderBufferPtrを呼んで適切なポインタを得る必要があります。ポインタが得られれば、あとはそのポインタを GL.IssuePluginEventの直前にプラグイン側に引き渡せば C#側の準備は終わりです。

RenderBuffer.GetNativeRenderBufferPtrで得られたポインタも実際のグラフィックス API のオブジェクトを参照している訳ではないため、プラグイン側で適切な関数を呼んでオブジェクトを得る必要があります。Vulkan を使用している場合は、IUnityGraphicsVulkan::access_render_buffer_textureもしくは IUnityGraphicsVulkan::access_render_buffer_resolve_textureを呼び出すことで、描画ターゲットとなっている VkImageオブジェクトを取得できます。これらの関数で同時にイメージレイアウト遷移などのパイプラインバリア操作も行えるため、適切な設定を渡したのち得られた VkImageをプラグイン側の VkSwapchainから得られた VkImageに vkCmdBlitImageで転送します。コマンドの積み込みは、今回は Unity の描画に割り込む訳ではないので自前で用意した VkCommandBufferに積んで、Unity 側から提供された VkQueueを通して送っています。

2.4 おわりに

今回は Unity のレンダリング周りに横槍を入れてみるお話でした。Screen Casting のデモでは、うまく応用すれば Unity 単体では難しい「ネット越しに相手の画面が見える」みたいな演出もできそうで、ネイティブプラグインの可能性は考えているよりかなりひろいと感じました。

第 3 章

Unity 2D Animation でキャラクターを動かす

Kinuko MIZUSAWA

普段の仕事ではクライアントエンジニアとしてビルドのマシンやジョブの管理、OS ネイティブ周りの機能を担当しています。ですが今回は久し振りにゲーム開発っぽいネタで行きたいと思います。Unity[*1]でオリジナル 2D キャラを動かします。

3.1 Unity 2019.2 で改善された 2D Animation 機能のワークフロー

2019 年 7 月にリリースされた Unity 2019.2 より、2D Animation 機能[*2]のスプライト画像として利用できる画像のフォーマットが増えました。それまでは PNG 形式などの一枚のスプライト画像だけでしたが、レイヤー構造を持った PSB ファイルも使えるようになっています。その結果、キャラクターの頭や腕、足、衣装などを個別のレイヤーとして持たせてキャラクターの IK アニメーションを実装することができるだけではなく、その IK アニメーションやリギングの情報を保持したまま、頭や腕、足、衣装などのパーツレイヤーをまるごと別のレイヤーに置き換えることができるようになりました。

このように、PSB ファイルのサポートによって 2D Animation 機能のワークフローが改善されています。実際に改善されたワークフローについては、Unity 公式ページの紹介動画[*3]を見ることですこし体感することができます。

この紹介動画では、キャラクターが手に持っているアイテムを Unity の Inspector で置

[*1] Unity 公式ページ `https://unity.com`

[*2] Unity 2D Animation 公式マニュアル `https://docs.unity3d.com/Packages/com.unity.2d.animation@3.0/manual/index.html`

[*3] Unity 公式ページでの 2D Animation のワークフロー改善紹介動画 `https://unity.com/ja/releases/2019-2/artist-and-designer-tools#2d-animation-swappable-sprites`

き換えても問題なくアニメーションが動いている様子が分かります。

ワークフローが改善される前の、一枚の画像に対してアニメーションを付ける形で同じことをする場合、アイテムの数だけキャラクターの画像を作り、一枚一枚にアニメーションを付けることを繰り返す必要がありましたが、2019.2 の改善によってそのような作業が不要になりました。

Unity 2D Animation の機能は Unity の Package Manager で提供されているパッケージをインポートすることで利用できます。

また Unity の公式サンプルとして、いくつかのキャラクターのアニメーションを実装したシーンファイルが提供されており、キャラクターの PSB ファイルやアニメーションファイルの中身を実際に動かしながら確認することができます。

今回はこのパッケージを使って 2D キャラにボーンをつけて動かして行こうと思います。

3.2 Unity の公式サンプル

実装を進めていく上では、Unity 公式のマニュアルやサンプルが参考になります。2019.2 用の 2D Animation のサンプルは次の URL で公開されています。

- `https://github.com/Unity-Technologies/2d-animation-v2-samples`

ちなみに、2D Animation の初期バージョン（Unity 2018.1 以降で利用可能）のサンプルも次の URL で公開されているので、差分が気になる方は見てみても面白いかもしれません。

- `https://github.com/Unity-Technologies/2d-animation-samples`

3.3 実装するキャラクターについて

▲図 3.1　KLab Tech Book Vol. 4 で登場したカナちゃんこと糸繰叶ちゃん

アニメーションを実装するキャラは KLab Tech Book Vol. 4[*4]で登場した糸繰叶ちゃんです。ロボや機械の体調が分かる機械オイルソムリエであり、色んなロボたちを作ることができる女の子。それでいてロボたちを作るために色んなパーツを買ったりするために常に金欠、しかもネーミングの縁起が悪いという愛されキャラです。

▲図 3.2　今回は直接登場しませんが... カナちゃんの相棒、クラッシュくん

[*4] KLab Tech Book Vol.4 http://klabgames.tech.blog.jp.klab.com/techbook/KLabTechBook_Vol4.pdf

カナちゃんは、Vol. 4 では表紙を飾るほか、ポストカードでも名誉を勝ち取ったシチュエーションで満面の笑みを見せてくれています。

▲図 3.3　糸繰叶ちゃんの参考資料：表紙、ポストカード、初期デザイン案

3.4　実装するゲームについて

糸繰叶ちゃんが登場する 2D ゲームとして、今回は横スクロールアクションゲームでアニメーションを実装してみます。

横スクロールアクションなので、次の動きは作っておきたいところです。

- 何もしてない時
- 歩いている時
- 走っている時
- ジャンプしている時
- 着地している時
- ロボに乗っている時
- ロボから降りている時

ただ今回は作業時間の都合上、走っている時のアニメーションに絞って実装します。

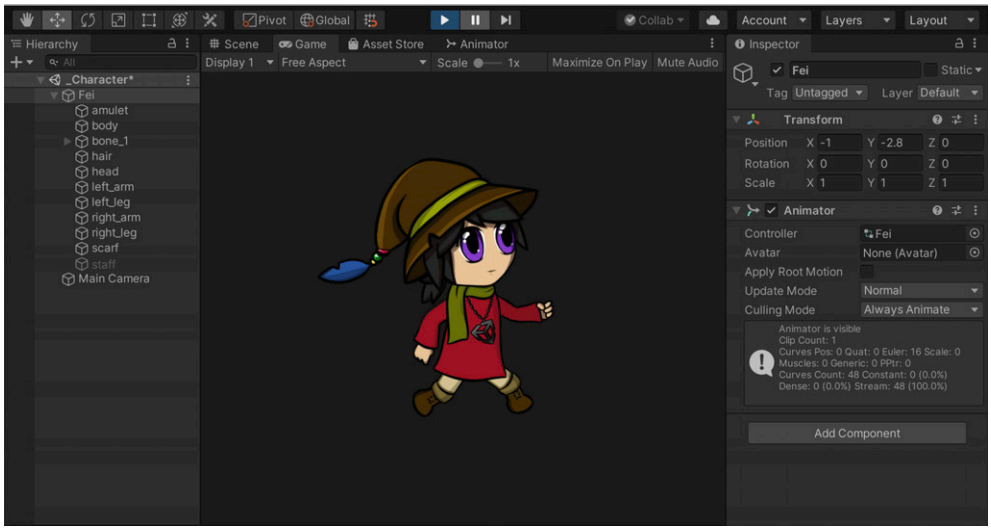
3.5　アニメーションの実装

実際に実装を進めて行きます。

PSB データの用意

あらかじめキャラクターのパーツごとにレイヤー分けしておいた PSB データを用意しておきます。

今回は 2D Animation のサンプルファイルにある Fei というキャラの PSB ファイルを参考にしてペイントソフト[*5]を利用して作成しました。



▲図 3.4　Unity のサンプルファイルにあるフェイ

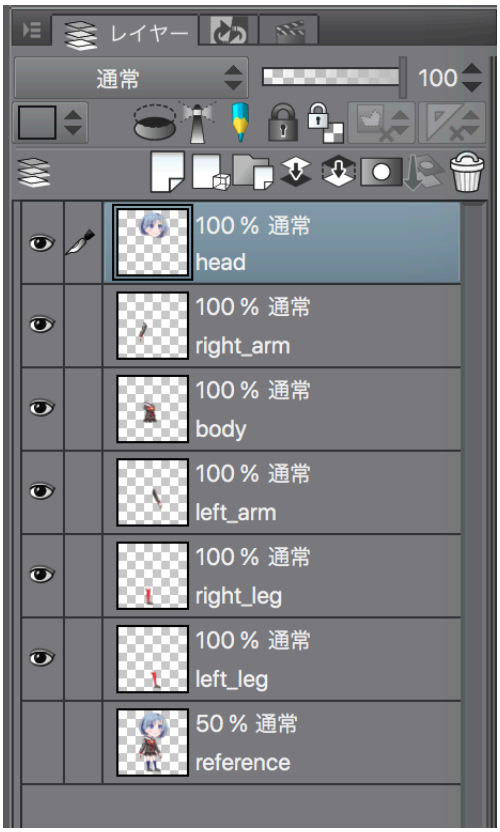
このフェイというキャラの PSB ファイルは、Unity エディタのシーンに配置してそのまま実行すると、帽子をかぶった魔導師風のキャラクターが手と足を前後させるアニメーションがループ再生されます。

まずは自分で用意した PSB ファイルで動かして 2D Animation の機能やキャラクターの動きなどを確認してみたかったので、最初は一枚のキャライラストをペイントツールで一枚レイヤー統合済みの画像として描き切ってからざっくりと **頭**（head）、**胴体**（body）、**両腕**（right_arm, left_arm）、**両足**（right_legs, left_legs）といったパーツで選択範囲を作成してイラストを分割し、それを元にスカートの中にあって見えない足の部分や、肩や腕で隠れてしまう胴体の部分などを描き足すようにしています。

[*5] Clip Studio `https://www.clipstudio.net/`

こうすることで、アニメーションで腕を前後に振る、足を上げ下げするといったことをさせても、描き足りない部分が出て不自然に見えることがなくなります。

そのときのレイヤー構成は下記です。



▲図 3.5　レイヤー構成

各レイヤーはそれぞれ次のような内容としています。

reference レイヤー

キャラクターの全体図が分かる参照レイヤー

head レイヤー

キャラクターの頭部分のみ描いたレイヤー

body レイヤー

キャラクターの胴体部分のみ描いたレイヤー

right_arm レイヤー

キャラクターの右腕部分のみ描いたレイヤー

left_arm レイヤー

キャラクターの左腕部分のみ描いたレイヤー

right_legs レイヤー

キャラクターの右足部分のみ描いたレイヤー

left_legs レイヤー

キャラクターの左足部分のみ描いたレイヤー

腕や脚などのレイヤーはアニメーションされることを想定して、スカートや肩周りの内側の見えない部分もすこし多めに描いておきます。

あとは一点注意点として、2D Animation でキャラクターなどのリグを組みアニメーションさせる場合は **PSB ファイルの最上位に** アニメーションさせたいキャラクターのパーツレイヤーを配置させないとうまく動かないので注意が必要です。

もしキャラクターのパーツレイヤーを PSB ファイルの最上位に配置させないと、後述するボーンやジオメトリなどのセットアップ作業を保存したタイミングでキャラクターのボーンやスプライトの位置がバラバラになってしまい、PSB ファイルの内容がおかしくなります。そして、一度このような状態になってしまうと、PSB ファイルをインポートするあたりから作業がやり直しになってしまいます。

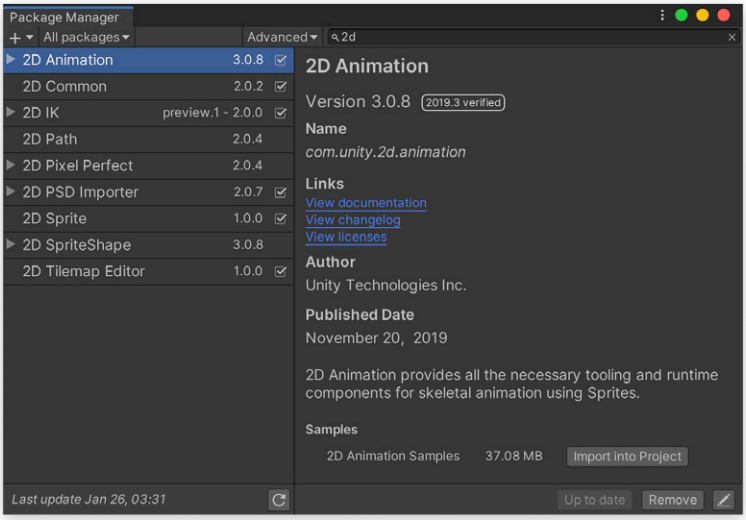
Unity Package のインストール

Unity Package をインストールします。インストールは Unity のメニューの Window > Package Manager から行います。

インストールするべきパッケージは下記です。すでにインストール済みの場合は再インストールする必要はありません。

- 2D Animation
- 2D Common
- 2D IK
- 2D PSD importer
- 2D Sprite

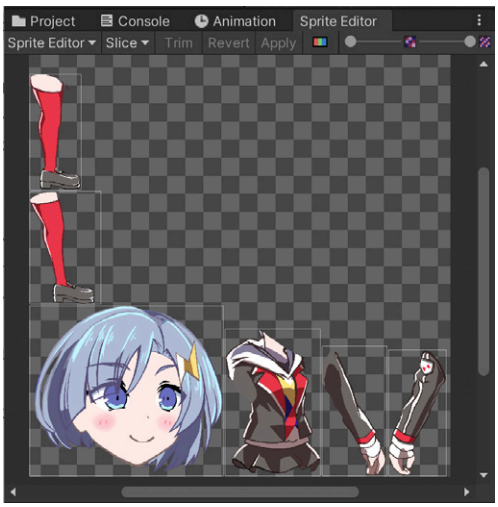
たとえば 2D Animation のパッケージは次のような画面からインストールすることができます。



▲図 3.6　2D Animation のパッケージ

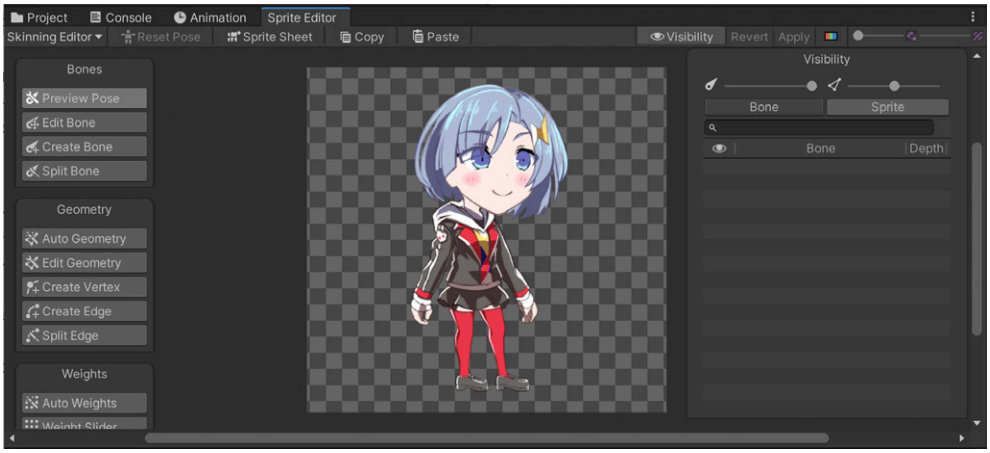
PSB データ読み込み

プロジェクトの適切なディレクトリに PSB データを入れておいた後は、Unity のメニューから Window > 2D > Sprite Editor を選択します。



▲図 3.7　Sprite Editor を起動したところ

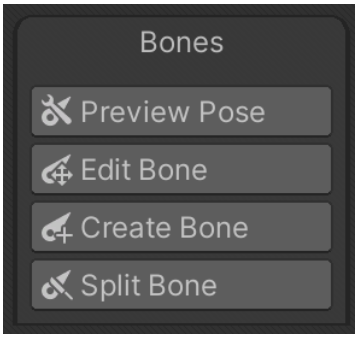
選択すると、エディタのウィンドウが起動します。以降はプロジェクトウィンドウで PSB ファイルを選択すると Sprite Editor 上の表示が更新され、選択した PSB ファイルに対してアニメーションの実装に必要なボーンやジオメトリ、リグの編集ができるようになります。



▲図 3.8　キャラクターのアニメーション設定（リギング）をする Skin Editor を起動したところ

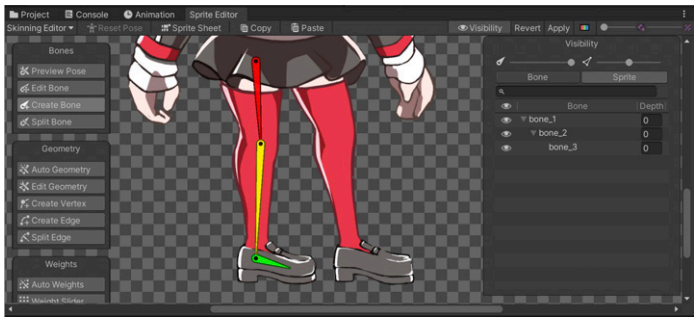
キャラクターのボーン設定

キャラクターの関節、曲げたいポイントを意識しながらボーンをつけて行きます。
ボーンをつけていく際は、Skinning Editor 左の以下のようなメニューを使用します。



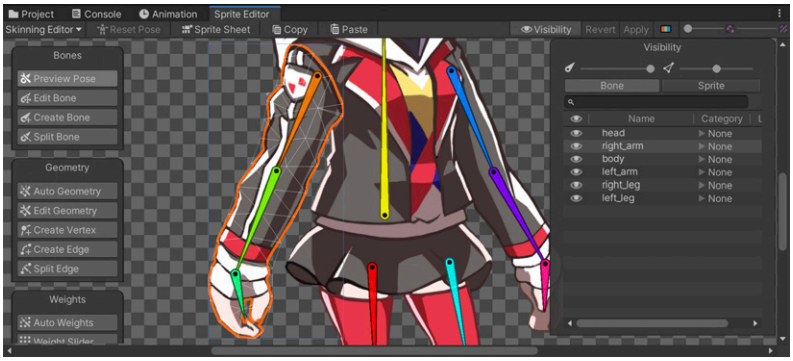
▲図 3.9　ボーンを設定するためのメニュー

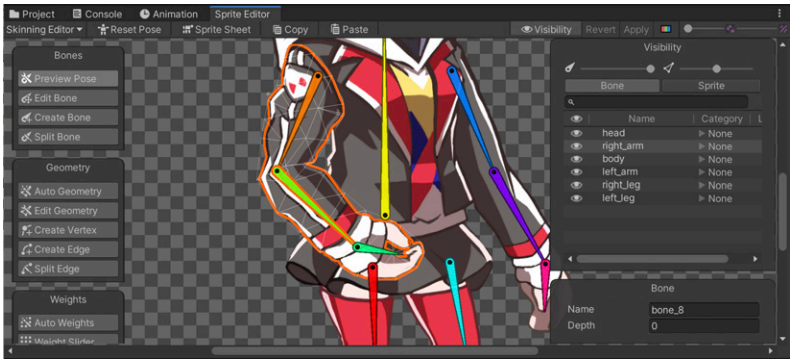
ボーンをつけていく時は、最初に Create Bone ボタンを押下してモードを変更します。配置はボーンの始点、太い方が回転の軸になるように配置していきます。



▲図 3.10　Skinning Editor で Create Bone を選択し、ボーンを追加したところ

　ボーンをつけただけではボーンに合わせて腕や足が追従して曲がってくれる事はありませんが、後述する設定をきちんと進めていくと、以下のようにきちんと曲がってくれます。



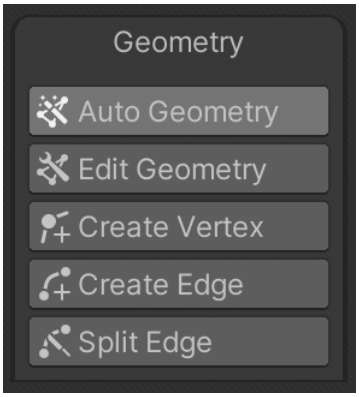


　たとえば足のボーンを付ける場合は足の付け根からボーンをつけていきます。膝のあたりでいったんボーンの終点をおき、またくるぶしのあたりで終点をおき、つま先のところで終点を置いた後に Ctrl+Z でボーンの追加を終了します。

ボーンの追加をした後は、Edit Bone ボタンを押下するとボーンの位置を微調整することができます。

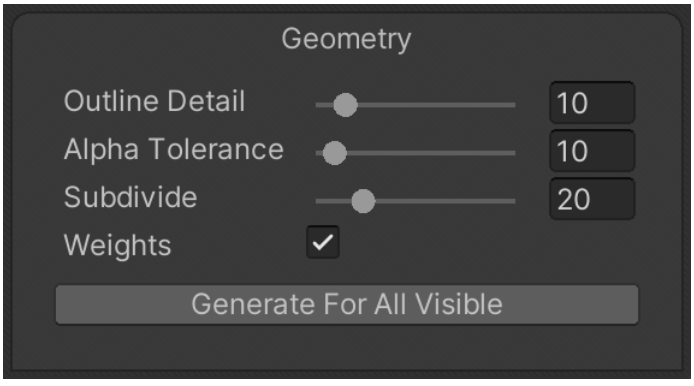
キャラクターのジオメトリ設定

ボーンを付けたら、スプライトを動かすためのジオメトリ（頂点）の作成を行います。
頂点の作成を行う際は、Skinning Editor 左の以下のようなメニューを使用します。



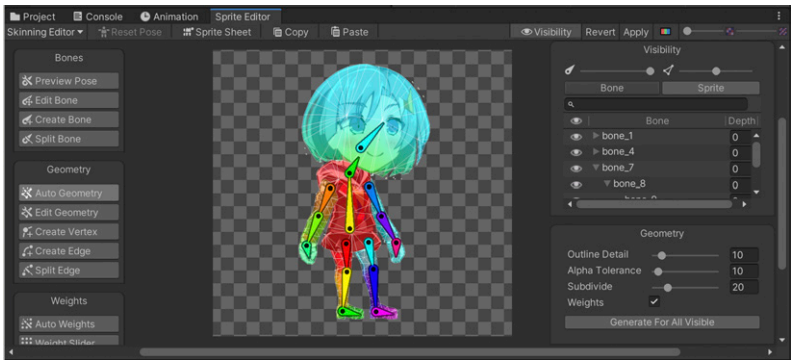
▲図 3.11　ジオメトリを設定するためのメニュー

ひとまずは Auto Geometry を選択して、Skinning Editor 右の Generate For All Visible ボタンを押してジオメトリをざっくり作ってしまいます。



▲図 3.12　ジオメトリを生成するためのメニュー

ジオメトリを作ると、PSB データの輪郭に合わせて大まかな頂点が作られます。



▲図 3.13　ジオメトリの作成が終わった状態の表示。ポリゴンで分割されたスプライトが表示される

ひとまずはこれで進めてみます。

キャラクターのウェイト設定

次に、キャラクターの曲げたいポイント、曲げたくないポイントを意識しながらウェイトをつけていきます。

ウェイトをつける際は、Skinning Editor 左下の以下のようなメニューを使用します。

▲図 3.14　ウェイトをつけるためのメニュー

先ほど、ジオメトリを自動生成した時点で、キャラクター全体にまだらに色が付いていることは確認できていたと思います。このように、ジオメトリを自動生成した時点でも、ざっくりとはウェイトをつけてくれているので、これでひとまずアニメーションの動きを確認することはできます。

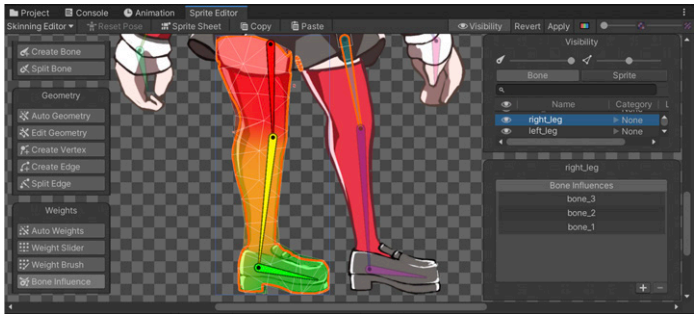
キャラクターのセットアップ確認

ここまでだいたいできたら、都度プレビューで動きを確認して調整していきます。Preview Pose ボタンを押すと、ボーンの終端を動かして回転させたり、終端以外を動かして位置を移動させたりして、アニメーションのセットアップの調子を確認することができます。Reset Pose ボタンを押すと、Preview Pose 中につけたポーズを初期状態に戻すことができます。Preview Pose ボタンを有効にした状態で色々な動きを確認しても、Reset Pose ボタンを押せば元の状態に戻せるというわけです。

もし、ここで二の腕の肘部分をドラッグしたのに、手首が連動して移動してくれない、ということがあれば Bone Influence ボタンを押して、スプライトごとにどのボーンが追従して動くかの設定をしましょう。

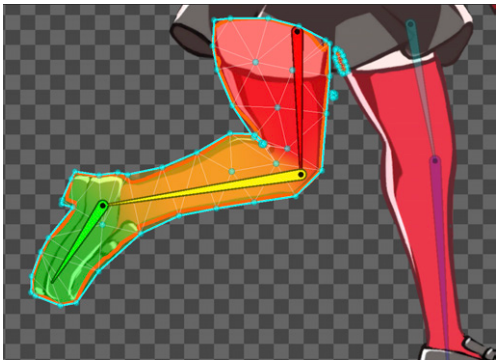
Bone Influence ボタンを押してボーンの追従設定を編集するモードになった後は、任意のボーンを選択すると右下の Visibility のパネルにスプライトに追従するボーンの一覧が表示されるようになります。ここで、スプライトの終点になるボーンの方から順番に上から表示されるように設定します。

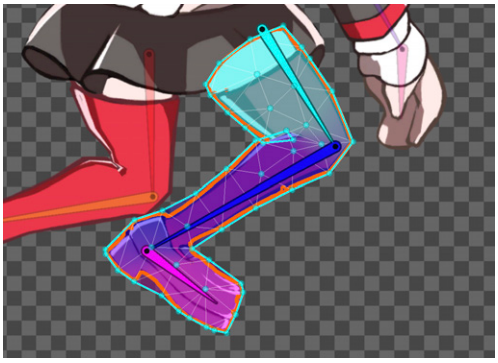
たとえば、右足（right_legs）なら次のようになります。



▲図 3.15　right_legs の Bone Influence の状態

ボーンやジオメトリの数や位置調整、ウェイトの調整をしつつ、各パーツの動きを確認していきます。



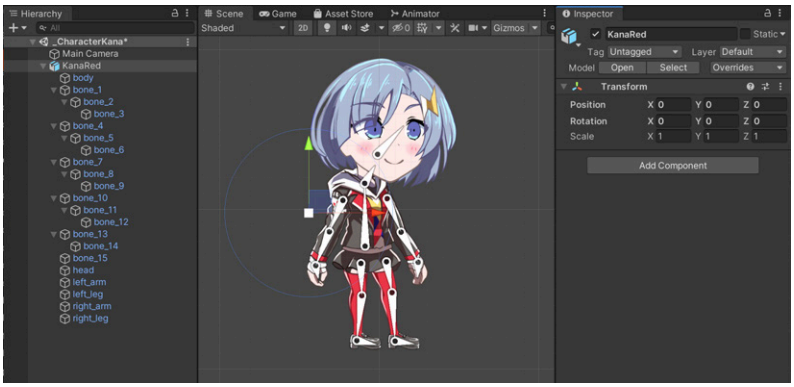


ここまでキャラクターのセットアップができたら、Apply を押してセットアップを保存します。

キャラクターのアニメーション実装する前に

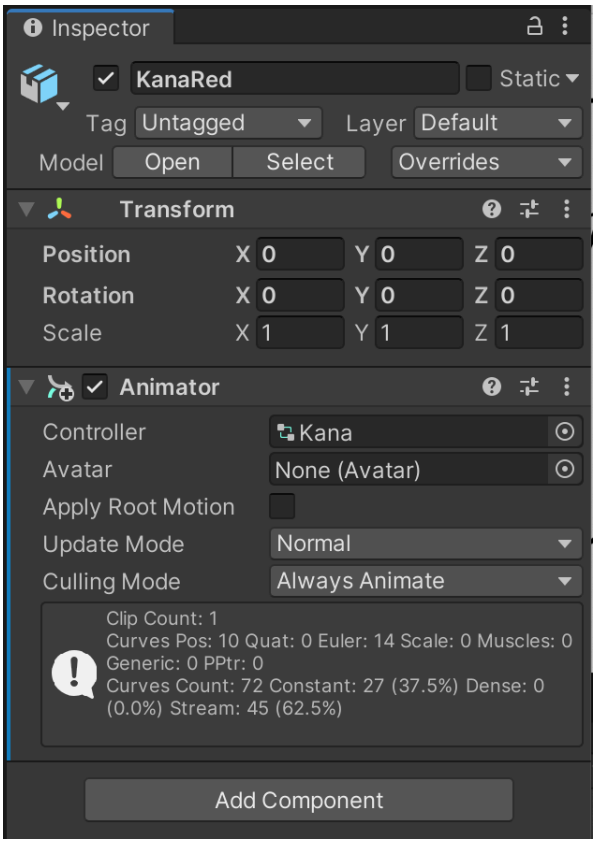
セットアップが終わったら、ついにアニメーションをつけていきます。
今回はわかりやすい動きとして走るモーションをつけていきます。

まず、シーンに作成したキャラクターの PSB ファイルをドラッグアンドドロップして配置します。



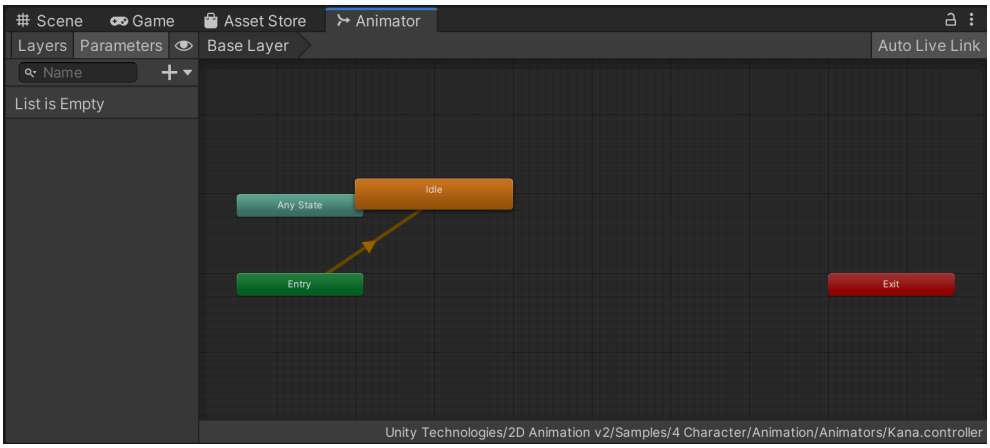
▲図 3.16　シーンにキャラクターを配置した状態

配置した後、PSB ファイル名と同じルートとなるオブジェクトに対して、Add Component で Animator を追加してキャラクターのアニメーションの状態遷移を管理するためのコントローラーファイルを追加作成します。



▲図 3.17　キャラクターのインスペクタでコントローラーがアタッチした状態

ここまでできたら、アニメーションのステートで、Entry ステートから遷移した先のステートとして Run ステートを作ります。

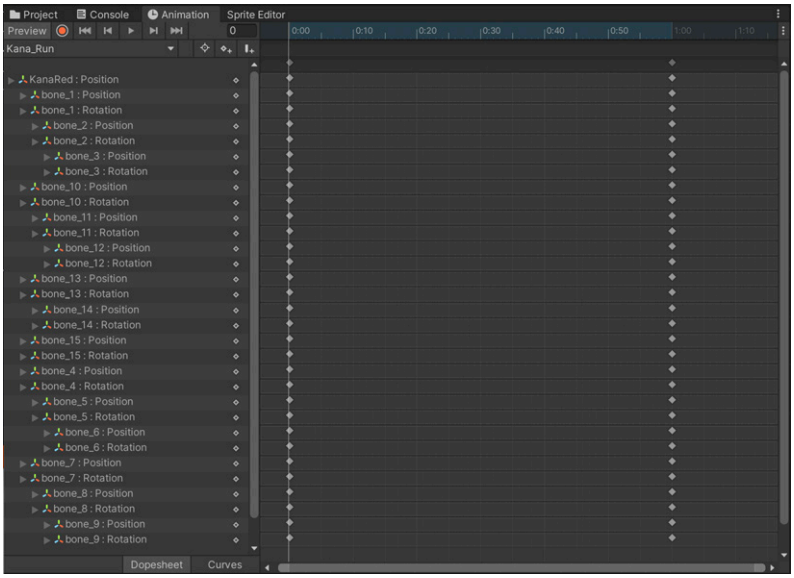


▲図 3.18　キャラクターのアニメーションステート

この Run ステートを右クリックしてアニメーションの動きを定義するクリップを作成します。

キャラクターのアニメーションをいざ実装

クリップを作成すると、タイムラインが表示され、動きをつけていくことができます。

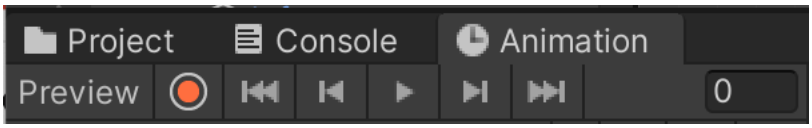


▲図 3.19　アニメーションのタイムラインエディタ

今回はボーンの位置と回転角だけでアニメーションをつけていくため、最初にすべての

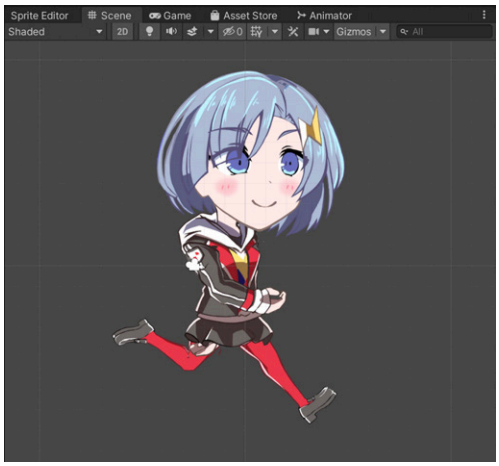
ボーンの Transform の Position と Rotation を Add Property しておきます。

動きをつけていく時は、ウィンドウ左上あたりにある、テレビの録画ボタンのような赤い丸ボタンを押します。これを押すと、フレームごとにポーズをつけてフレーム間の動きを補間させることが出来るようになります。



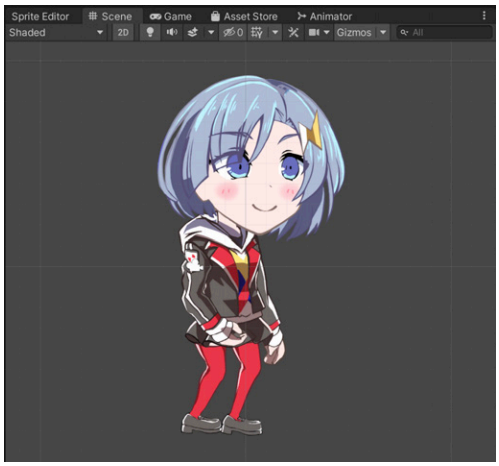
▲図 3.20　アニメーションのポーズ付けに役立つ録画ボタンなどが並ぶ UI

今回は走るモーションをつけたいので、最初の 0 フレーム目でキーフレームを追加し、めいっぱい足をストライドさせた状態の見た目を作ります。赤い丸ボタンが有効になっていれば、Skinning Editor でセットアップをしていた時と同じようにボーンの終端を動かせば始点を中心に回転して追従するボーンとスプライトが動いてくれます。最初の 0 フレーム目の見た目は次のような状態にしました。



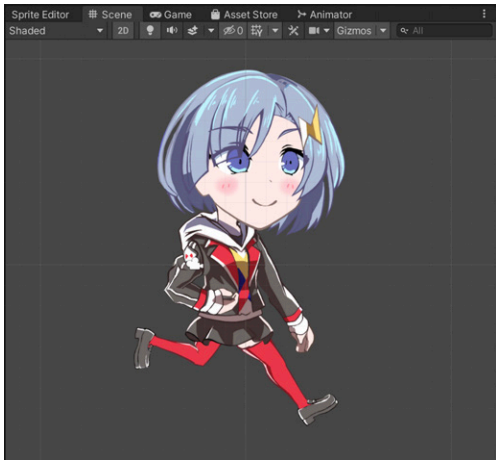
▲図 3.21　走っているアニメーションの 0 フレーム目の状態

その後、30 フレーム目でまたキーフレームを追加し、だいたい両足が重なるタイミングの見た目を作ります。最初の 30 フレーム目の見た目は次のような状態にしました。



▲図 3.22　走っているアニメーションの 30 フレーム目の状態

その後、60 フレーム目でまたキーフレームを追加し、一番最初のフレームとは逆の足が前に・後ろに来ている状態の見た目を作ります。最初の 60 フレーム目の見た目は次のような状態にしました。



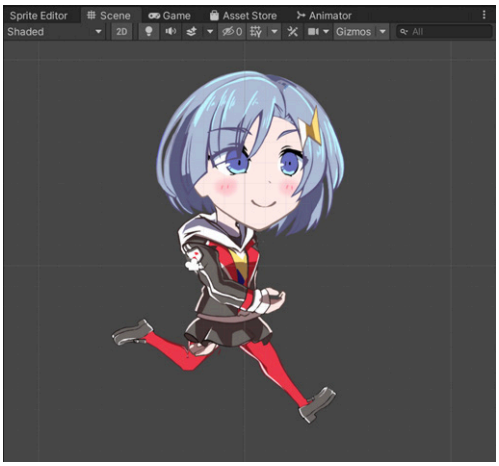
▲図 3.23　走っているアニメーションの 60 フレーム目の状態

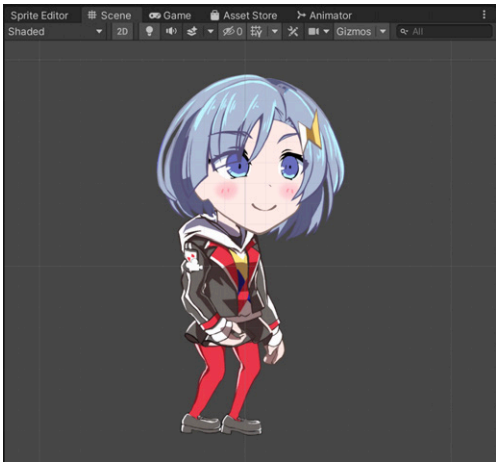
3.6　アニメーションの実装結果

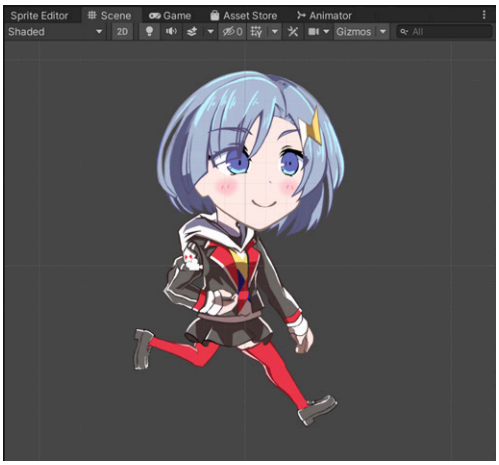
こうして作ったアニメーションを、Animation ウィンドウの再生ボタンを押すあるいはシーンを実行させて確認します。

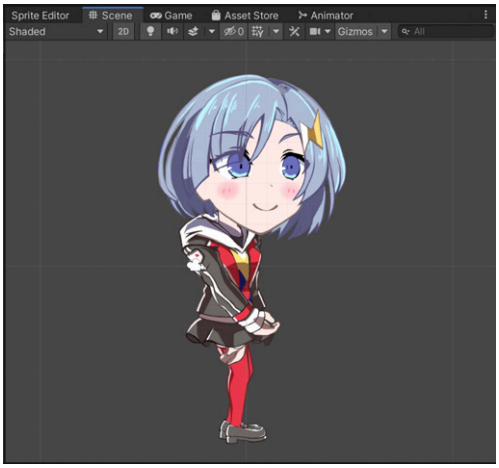
キーフレームとキーフレームの間のフレームは、自動的に補間されなめらかなアニメー

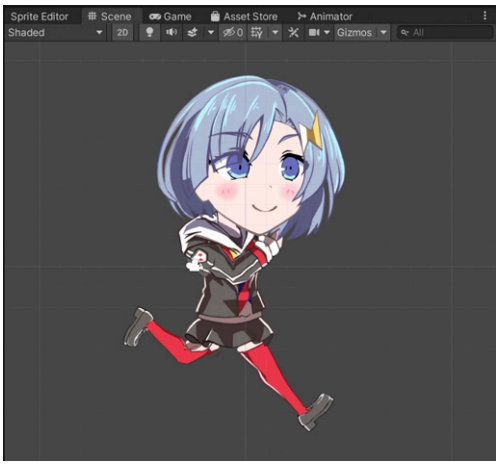
ションになっていることが確認できます。






Sprite Editor
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
All


Sprite Editor
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
All


Sprite Editor
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
All

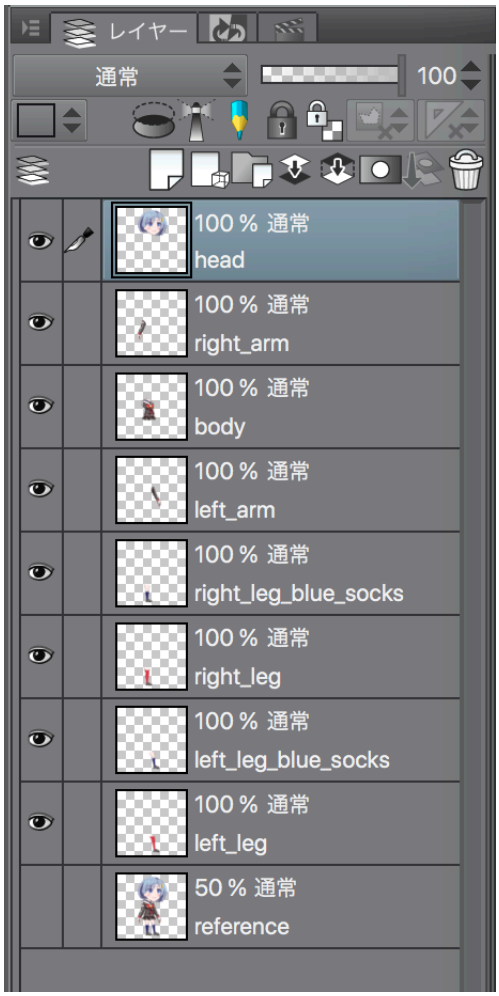
ここまで作ったやり方を進めていけば、歩いている時、ジャンプしている時、ロボに乗る時、ロボから降りている時などのアニメーションもそれぞれ作ることができます。

3.7 実装したキャラクターのパーツ入れ替え

さて、ここまで Unity 2D Animation で実装してきたアニメーションですが、最初に紹介したように、Unity 2019.2 でパーツの入れ替え作業が簡単になっているとのことでした。最後にちょっとだけこの機能を使ってみましょう。

靴下の色変えレイヤーの追加

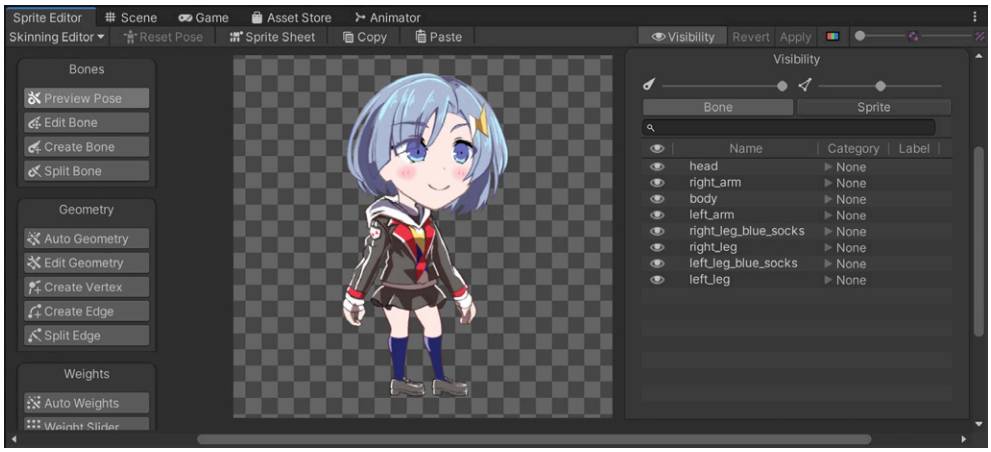
今回は、靴下の色変えをできるようにしたいと思います。アニメーションの実装を行なった PSB ファイルをもう一度ペイントソフトで開き、元々描いていた**赤い靴下の**足のパーツとは別のレイヤーを作成して**紺の靴下の**足のパーツのレイヤーを用意します。



▲図 3.24　ペイントソフト上で靴下を変えた足のパーツを追加した時のレイヤー構成

色変えレイヤーの PSB ファイルで更新

レイヤーを追加した PSB ファイルを Unity の Skinning Editor で開くと、Sprite の種類が増えていることがわかります。

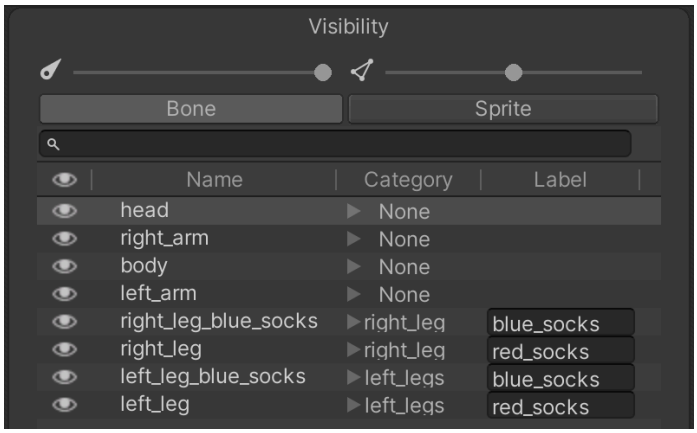


▲図 3.25　Sprite に追加した足のパーツのレイヤーがあることを確認した画面

Unity 上で PSB ファイルのスプライトのカテゴリを設定

Sprite が増えているので、靴下の色が異なるそれぞれの足のパーツを明示的に切り替えられるようにするためにカテゴリとラベルを指定してあげます。カテゴリには左足なら左足、右足なら右足といったようにデザインを切り替えたいパーツの種類を示す名前を入れ、ラベルにはカテゴリの中で識別できるような名前を入れます。

今回は、次のように右足、左足を設定しました。

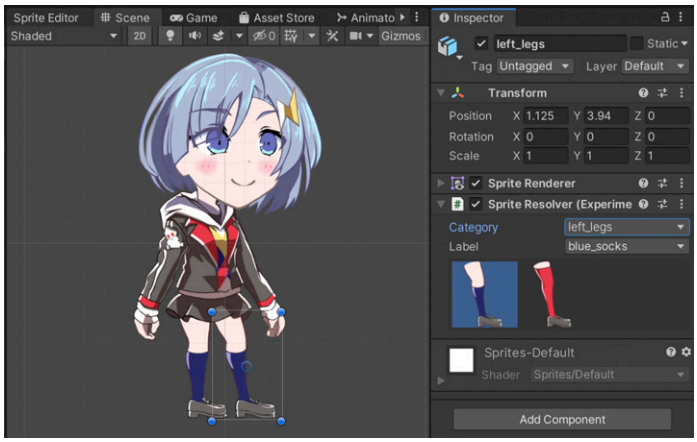


▲図 3.26　Category と Label の設定

スプライトの切り替え

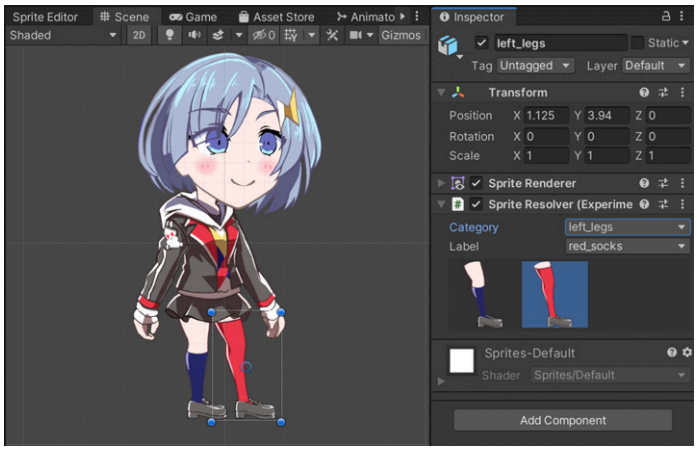
ここまで設定が終わったら、Inspector でパーツの入れ替えが出来るようになっています。

Hierarchy 上のルートオブジェクトを辿り、左足のゲームオブジェクトを選択して Inspector を見てみると、Sprite Resolver というコンポーネントが付いており、そのコンポーネントの項目上に左足のカテゴリに設定しておいた赤と紺それぞれの左足のスプライトが表示されていることが分かります。



▲図 3.27　Sprite Resolver で左足に紺の靴下を選択している状態

このスプライトの選択を変更してみると、即座にシーン上の表示が切り替わる事が確認出来ます。



▲図 3.28　Sprite Resolver で左足に赤の靴下を選択している状態

今回は Inspector で直接変更していますが、シーン実行時にボタン押下時のイベントでパーツの入れ替えを行なっている公式のサンプルもありました。ですので、プログラムからも変更できるようです。

3.8　まとめ

ここまで、KLab Tech Book Vol.4 のキャラクターを用いて、2D ゲームのキャラアニメーション作成について紹介してきました。

見たところ 2020 年 2 月現在の 2D Animation のバージョンでは、Sprite Skinning Editor で設定するボーンの情報[*6]を異なる PSB ファイル間においてコピー＆ペーストすることはできないようでした。ですので、リグの情報を複数のスプライトで共用したい場合は、現時点では関連するスプライトのレイヤーをひとつのファイルにまとめておく必要がありそうです。

今回は自作のキャラクターのアニメーション中などにパーツの入れ替えをするところまでは紹介しませんでしたが、公式のサンプルで似たようなことを行なっているファイルもありますし、そのほかにも今回紹介していないスプライトの入れ替え方法なども載っているので、もっと詳細を追ってみたいという方はそちらを追ってみると良いと思います。

最後になりますが、Unity の 2D Animation でしっかりと動きをつけていくとなると、やはり相応のスキルや時間が必要になるとは思います。しかしこうした機能でかわいい 2D キャラのアニメーションを作り、隙間時間で遊んでもらえるカジュアルゲームを作ってみるのも楽しそうです。本記事の内容が、読者のみなさまのお役に立てればさいわいです。

[*6] ボーンの位置情報などは、スプライトのカテゴリ、ラベルなどの情報と一緒に、PSB ファイルのメタファイル（.meta）に含まれています

第 4 章

Arduino UNO を USB 接続の HID デバイスにする

Toshifumi Umezawa / @umezawa-to

4.1 はじめに

人は誰しもゲームパッドを自作したくなるときが来ます。これをマイコンの Arduino シリーズを使って実現する場合、Arduino Leonardo という USB 接続機器製作向けの種類が適任ですが、汎用的な用途の Arduino UNO しか手持ちがないということもあると思います。

今回は、この一番ノーマルな Arduino UNO を HID のゲームパッド化し、必要数のボタンとアナログ入力の数に変更し、物理的な操作を入力として扱う方法について書いていきます。操作の種類については、ボタンのように**押す**操作だけだと独自性がないので、今回は**回転**の操作も認識できるようにしてみます。これによって、ボリューム調節ツマミのようなものを被せて手で掴んで操作させたり、ターンテーブルを取り付けてスクラッチ操作をさせるなど、特殊な操作を行うゲームパッドが作れます。

章の前半で前提知識をざっくりと説明したのち、後半で実際のプログラムなどを説明します。

4.2 HID レポートディスクリプタ

HID（Human Interface Device）とは、人が制御する入力装置を扱う規格です。

自作の特殊なデバイスを PC などで使うには、デバイスドライバを用意したりアプリ側でロジックを知っておいたりなどをする必要がありますが、HID に準拠したデバイスとして製作することで、OS 標準の汎用 HID デバイスドライバで認識され、アプリ側でも汎用の関数を使ってマウスやゲームパッド等の入力を受け取ることができます。

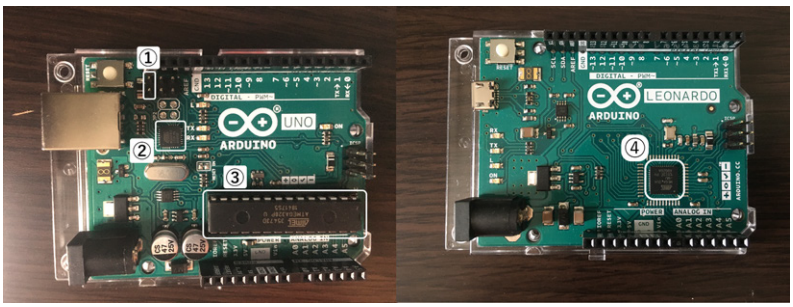
HID に準拠したデバイスでは、接続時にデバイスの情報としてさまざまなデータを渡します。その中で今回特に重要なのは、**レポートディスクリプタ**と呼ばれるデータです。デバイスの状態は**レポート**という単位で送信されますが、レポートディスクリプタはこのレポートの構造を示すものです。

たとえば、デバイス側が `[1A,2B,09,00]` というバイト列を流したとします。受け取り手が事前に受け取ったデバイス情報のレポートディスクリプタを読み込み、「1byte で表されるアナログ情報 2 つと、2byte で表されるボタン 16 個分の情報が渡される」ことを知っていれば、このバイト列を「アナログ 1 入力: `1A`, アナログ 2 入力: `2B`, ボタン入力: `00001001 00000000`」というように正しく取り出してデバイスの状態を認識することができます。

4.3 DFU モード

Arduino では**スケッチ**という C 言語のプログラムを書き込んで回路の制御を行えます。**DFU モード**は、スケッチの記憶領域とは別の場所にあるマイコンのファームウェアを書き換えるモードです。

DFU モードで起動するには、図 4.1 の①の 2 ピンを接続した状態で電源を入れるという操作が必要です。また、新しいファームウェアの書き込みには dfu-programmer という実行ファイルを使用します。dfu-programmer はプロジェクトのサイトからダウンロードできます[*1]。



▲図 4.1　Arduino UNO(左) と Leonardo(右)

デフォルトでは「COM ポートを通して PC と通信し、書き込まれたスケッチのプログラムを実行するファームウェア」が入っており、USB で接続すると「Arduino UNO」というデバイスとして認識されます。今回はそもそもゲームパッドとして認識してほしいので、それ用のファームウェアを書き込むことにします。

[*1] `http://dfu-programmer.github.io/` からリンクされた SourceForge よりダウンロード可能

ファームウェアとスケッチを格納するマイコン

図 4.1 の②がファームウェアを動かす ATmega16U2、③がスケッチを動かす ATmega328P です。UNO は 2 つのチップを使っていますが、Leonardo は④の ATmega32U4 の 1 つだけでスケッチの動作と PC との通信を行っています。
UNO に使われている ATmega328P は簡単な電子工作レベルにおいても Arduino を介さず直接回路実装をすることがあり、Arduino に搭載されたものを引き抜いて使用する人もいます。

4.4 hex ファイルと LUFA

hex ファイルはマイコンを制御する情報が書かれたファイルです。今回 DFU モードで書き込むのはこの形式のファイルです。

LUFA は USB 接続のマイコンのファームウェアを作るためのフレームワークです。LUFA のプロジェクトとして作成したものをビルドすることで、マイコンを USB 機器として動作させるための hex ファイルを生成することができます。

4.5 実践

ここまで説明した前提知識を踏まえて、実際に Arduino UNO のデバイスを変更してみます。フレームワークがあるとはいえイチから作るのは大変なので、すでにある実装例をもとに改変して作ることにします。

1byte アナログ入力 6 個とボタン 32 個を実装した arduino-big-joystick という LUFA のプロジェクトを作っている方がいた[*2]ので、このプロジェクトを少し書き換えて 1byte アナログ入力 2 個とボタン 16 個にしてみます。

4.6 ビルド環境

まずは LUFA のプロジェクトをビルドできる環境を用意します。avr-gcc コマンドとライブラリがあれば OK なので、必要なものを入れた簡単な Docker コンテナを作成します。ホストの PC からマウントされた arduino-usb ディレクトリの中の arduino-big-joystick を make して hex ファイルをビルドするだけのものです。

[*2] `https://github.com/harlequin-tech/arduino-usb.git`

▼リスト 4.1　Dockerfile

```
FROM ubuntu:18.04

RUN apt-get update
RUN apt-get -y install make gcc-avr avr-libc

RUN echo "cd arduino-usb/firmwares/arduino-big-joystick/ &&" >> task.sh &&
    echo "make &&" >> task.sh &&
    echo "cp arduino-big-joystick.hex ../out.hex &&" >> task.sh &&
    echo "make clean" >> task.sh &&

CMD ["/bin/bash", "task.sh"]
```

Dockerfile が用意できたら、プロジェクトを clone したうえでコンテナを使ってビルドします。

```
$ git clone https://github.com/harlequin-tech/arduino-usb.git
$ docker build -t joystick-hex-builder .
$ docker run --rm -v `pwd`/arduino-usb:/arduino-usb joystick-hex-builder
```

ビルドが終わると、arduino-usb/firmwares/に out.hex が書き出されます。

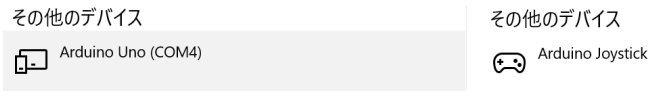
4.7　ファームウェアの書き込み

Arduino を DFU モードにした状態で、dfu-programmer を使って hex ファイルを書き込みます。atmega のドライバは dfu-programmer に同梱されているので、もし DFU モード中にマイコン名の atmega16u2 が表示されない場合は手動でドライバを指定してインストールしましょう。

実行ファイルと hex ファイルを準備し、DFU モード中の Arduino UNO を接続した状態になったら、次のようにコマンドを実行します。flash の引数に入れたい hex ファイルを指定します。

```
$ dfu-programmer atmega16u2 erase
$ dfu-programmer atmega16u2 flash Arduino-big-joystick.hex
$ dfu-programmer atmega16u2 reset
```

Arduino UNO を USB で差し込み直すと、ゲームパッドとしてドライバが設定されているのが確認できます。



▲図 4.2　Arduino UNO のデバイス表示 (左) と変更後 (右)

元の Arduino UNO に戻したい場合は、Arduino IDE に同梱されている hex ファイルを書き込みます[*3]。

4.8 レポートディスクリプタの変更

ここまででファームウェアのビルドと流し込みができるようになったので、今度はレポートディスクリプタを変更し、ボタン数などを変更してみます。

arduino-big-joystick はアナログ 2byte*8 とデジタル 1bit*40 の計 21byte のレポートを送信しますが、これをアナログ 1byte*2 とデジタル 1bit*16 の計 4byte にします。変更箇所はリスト 4.2 のとおりです（紙面の都合で改行や空白の位置を調整しています）。

▼リスト 4.2　descriptor_diff

```
--- a/firmwares/arduino-big-joystick/Arduino-big-joystick.c
+++ b/firmwares/arduino-big-joystick/Arduino-big-joystick.c
 USB_ClassInfo_HID_Device_t Joystick_HID_Interface = {
 /** Circular buffer to hold data from the serial port before it is sent to the
  host. */
 RingBuff_t USARTtoUSB_Buffer;

-USB_JoystickReport_Data_t joyReport = { {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0,
 0} };
+USB_JoystickReport_Data_t joyReport = { {0, 0}, {0, 0} };

 #define LED_ON_TICKS 2000      /* Number of ticks to leave LEDs on */
 volatile int led1_ticks = 0;
--- a/firmwares/arduino-big-joystick/Arduino-joystick.h
+++ b/firmwares/arduino-big-joystick/Arduino-joystick.h
  *  This mirrors the layout described to the host in the HID report descriptor
, in Descrip
tors.c.
  */
 typedef struct {
-    int16_t axis[8]; /**< Current absolute position axis 1-8, signed 16-bit in
teger */
-    uint8_t button[5];  /**< Bit mask of the currently pressed joystick button
s 1-40, 8 per byte */
+    int8_t axis[2]; /**< Current absolute position axis 1-2, signed 8-bit inte
ger */
+    uint8_t button[2];  /**< Bit mask of the currently pressed joystick button
s 1-16, 8 per byte */
 } USB_JoystickReport_Data_t;

 /* Macros: */
--- a/firmwares/arduino-big-joystick/Descriptors.c
+++ b/firmwares/arduino-big-joystick/Descriptors.c
 const USB_Descriptor_HIDReport_Datatype_t PROGMEM JoystickReport[] =
 {
        0x05, 0x01,          /* Usage Page (Generic Desktop)                 */
-       0x09, 0x04,          /* Usage (Joystick)                             */
+       0x09, 0x05,          /* Usage (GamePad)                              */

        0xa1, 0x01,          /* Collection (Application)                     */
        0x09, 0x01,          /*   Usage (Pointer)                            */

-       /* 8 axes, signed 16 bit resolution, range -32768 to 32767 (16 bytes) */
+       /* 2 axes, signed 8 bit resolution, range -256 to 255 (2 bytes)       */
        0xa1, 0x00,          /*   Collection (Physical)                      */
```

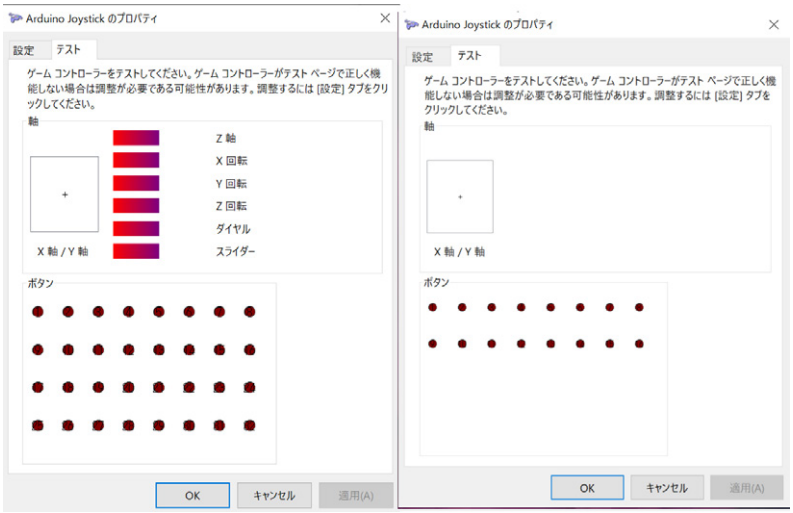
[*3] こちらの URL からも取得可能: https://github.com/arduino/ArduinoCore-avr/blob/master/firmwares/atmegaxxu2/UNO-dfu_and_usbserial_combined.hex

```
        0x05, 0x01,          /*     Usage Page (Generic Desktop)              */
        0x09, 0x30,          /*     Usage (X)                                 */
        0x09, 0x31,          /*     Usage (Y)                                 */
-       0x09, 0x32,          /*     Usage (Analog1)                           */
-       0x09, 0x33,          /*     Usage (Analog2)                           */
-       0x09, 0x34,          /*     Usage (Analog3)                           */
-       0x09, 0x35,          /*     Usage (Analog4)                           */
-       0x09, 0x36,          /*     Usage (Analog5)                           */
-       0x09, 0x37,          /*     Usage (Analog6)                           */
-       0x16, 0x00, 0x80,    /*     Logical Minimum (-32768)                  */
-       0x26, 0xff, 0x7f,    /*     Logical Maximum (32767)                   */
-       0x75, 16,            /*     Report Size (16)                          */
-       0x95, 8,             /*     Report Count (8)                          */
+       0x15, 0x80,          /*     Logical Minimum (-256)                    */
+       0x25, 0x7f,          /*     Logical Maximum (255)                     */
+       0x75, 8,             /*     Report Size (8)                           */
+       0x95, 2,             /*     Report Count (2)                          */
        0x81, 0x82,          /*     Input (Data, Variable, Absolute, Volatile)*/
        0xc0,                /*   End Collection                              */

-       /* 40 buttons, value 0=off, 1=on (5 bytes) */
+       /* 16 buttons, value 0=off, 1=on (2 bytes) */
        0x05, 0x09,          /*   Usage Page (Button)                         */
        0x19, 1,             /*     Usage Minimum (Button 1)                  */
-       0x29, 40,            /*     Usage Maximum (Button 40)                 */
+       0x29, 16,            /*     Usage Maximum (Button 16)                 */
        0x15, 0x00,          /*   Logical Minimum (0)                         */
        0x25, 0x01,          /*   Logical Maximum (1)                         */
        0x75, 1,             /*   Report Size (1)                             */
-       0x95, 40,            /*   Report Count (40)                           */
+       0x95, 16,            /*   Report Count (16)                           */
        0x81, 0x02,          /*   Input (Data, Variable, Absolute)            */
        0xc0                 /* End Collection                                */
 };
```

特にポイントとなるのが JoystickReportの値です。レポートディスクリプタの値は、このように byte 列で構造化された情報を直接指定します。ここで指定した内容に合わせて、他の箇所もデータサイズが合うように修正しました。

すべて変更が終わったら、ビルドしてでき上がった hex ファイルを Arduino UNO に書き込みます。変更前後のゲームパッドの状態は図 4.3 のように変わります。



▲図 4.3　変更前後のゲームパッド

4.9　スケッチの作成

ファームウェアの変更では USB 接続時の認識情報を変更しただけなので、実際に送るバイト列やその転送速度などはスケッチで準備します。気を付ける点として、ファームウェア書き込み後はジョイパッドなので、スケッチを書き込む機能はありません。適宜元のファームウェアに戻して書き込みます。

今回、ゲームパッドのボタン入力情報を出力するスケッチとしてリスト 4.3 を用意しました。3~10 番ピンのデジタル入力をゲームパッドの 1~8 番のボタンに割り当てています。

残りの 9~16 番のボタンにはフレームカウントのようなものを出力しておきます。

このスケッチを書き込んだ上で前述のファームウェアを書き込むことで、3~10 番ピンに入力されたときに対応するボタンが押された状態になります。また、ゲームパッドの 9~16 番のボタンは時間の経過とともに 8bit でのカウントが進む状態になっていると思います。

レポートの送信間隔を 16ms ごとに調整し、送信のタイミングで digitalRead しているため、送信の合間に押す・離すが行われた場合は認識されません。今回は説明の簡略化のため、あまり厳密に考えないことにしておきます。

▼リスト 4.3　sketch.ino

```
const int PIN_BUTTON_START = 3;
const int INTERVAL_MILLIS = 16;

typedef struct t_joyReport {
    uint8_t x;
```

```
    uint8_t y;
    uint16_t buttons;
} t_joyReport;

t_joyReport joyReport;
int pre_millis = 0;

void readButtons()
{
    joyReport.buttons = 0;
    for (int i = 0; i < 8; i++) {
        uint32_t button_input = digitalRead(PIN_BUTTON_START+i);
        joyReport.buttons |= (button_input<<i);
    }
}

uint16_t count = 0;

void sendJoyReport(struct t_joyReport *report)
{
    readButtons();
    count++;
    joyReport.buttons |= (count<<8);
    Serial.write((uint8_t *)report, sizeof(t_joyReport));
}

void setup()
{
    for (int i = 3; i <= 10; i++) {
        pinMode(i,INPUT_PULLUP);
    }
    Serial.begin(115200);
}

void loop()
{
    int new_millis = millis();
    if (new_millis - pre_millis > INTERVAL_MILLIS) {
        sendJoyReport(&joyReport);
        pre_millis = new_millis;
    }
}
```

4.10　スケッチの作成（アナログ回転情報）

初めに宣言したように、ツマミの回転をアナログ入力として認識できるように実装します。

今回、回転の認識に使用するのは「RES20D50-201-1」というロータリーエンコーダです。Amazon で 1000 円後半くらいで買えます。このパーツは 5V 電圧で動作し、クリックなしで滑らかに回転し、回転角によって 2 つの信号線から 5V 電圧のデジタル信号が出力されます。2 つの信号は、2bit で表すと [00,01,11,10] という固定順で繰り返すようになっており、変化したビットを見ることでどちらに回転したのかが分かるようになっています。

普通に購入すると短いケーブルが付いているだけなので、写真右のパーツは取り回しが良くなるようにコネクタを実装しています。

▲図 4.4　ロータリーエンコーダ RES20D50-201-1

このパーツの 5V,GND をそれぞれ接続したうえで、2 つの信号線を Arduino UNO の A0,A1 ピンに接続し、スケッチの適切な箇所にリスト 4.4 の内容を追記します。これにより、回転方向によって x 軸,y 軸が正の方向に移動するようになります。

ここで注意点がひとつあり、回転の認識をする readRotate を可能な限り細かく呼び出してあげる必要があります。信号の 2 値の組は 4 種類をループするため、たとえば「右に 1 つ分」と「左に 3 つ分」の信号は同じになります。右か左に 1 つ分の移動のみを検出する必要があるので、readRotate は HID のレポート送出よりも細かいタイミングで呼び出してあげます。

▼リスト 4.4　sketch.ino

```
const int PIN_SIG1 = 14;
const int PIN_SIG2 = 15;

int rot_arr[4] = {0,1,3,2};
int pre_a = 0;

void readRotate()
{
    int a1 = digitalRead(PIN_SIG1);
    int a2 = digitalRead(PIN_SIG2);
    int a = rot_arr[a1+a2*2];
    int a_diff = a - pre_a;
    if(a_diff==1||a_diff==-3) {
        joyReport.x++;
    }
    else if(a_diff==-1||a_diff==3) {
        joyReport.y++;
    }
    pre_a = a;
}

void setup()
{
    // 追記
    pinMode(15,INPUT_PULLUP);
    pinMode(16,INPUT_PULLUP);
}

void loop()
{
    // 追記
    readRotate();
}
```

RES20D50-201-1 の先端の取り扱い

ロータリーエンコーダの先端にはツマミ等を取り付けますが、ネジとして扱いたい場合は変換用の部品を噛ませる必要があります。電子パーツ店でロボ系の陳列（ギアやシャフトを売っているあたり）の中に「エンドネジ」というシャフト⇔ネジの変換パーツがあり、これの外径 6mm のものがちょうどはまるパーツになります。

▲図 4.5　エンドネジ

4.11　まとめ

このような実装により、必要数の物理ボタンおよび回転を認識する HID のゲームパッドを、Arduino UNO を使って作成することができました。また、スケッチで Serial.write している箇所をゲームパッドとして認識される無線モジュールへ送信するように書き換えることで、無線ゲームコントローラーを作成することもできます。ただし、無線モジュールは今回説明したようにファームウェアの書き換えができないため、モジュール内蔵の仕様に合わせたレポートを送出する程度の改変に留まると思います。

皆さんも Arduino UNO しかない状況になったら試してみてください。Arduino Leonardo など ATmega32U4 チップが使われているマイコンを使う方が断然手間が少ないので、お金があれば素直に Leonardo 買いましょう。

第 5 章

バーコードリーダーになろう — Code128 編

Daisuke Makiuchi / @makki_d

バーコード、読んでますか？

KLab Tech Book Vol.3「バーコードリーダーになろう」[*1]ではレジで読み取るような EAN 規格[*2]のバーコードの読み方を解説しました。お読みになった皆さんは、日々バーコードを肉眼で読んでいることと思います。

ところで、図 5.1 のような箱を見たことがきっとあることでしょう。

▲図 5.1　よく見かける箱

この箱に付けられているバーコード 3 種類を拡大したものが図 5.2 になります。これらのバーコードは EAN ではなく、すべて **Code128** という規格のバーコードです。

[*1] これまでの KLabTechBook の PDF を無料頒布しています。
`http://klabgames.tech.blog.jp.klab.com/archives/tbf08.html`

[*2] European Article Number。国コードが日本のものは JAN（Japanese Article Number）とも呼ばれます。

▲図 5.2　箱に付けられたバーコード

この章では、私達の身近にも使われているこの Code128 について解説します。読み方をマスターしましょう。

5.1　Code128 とは

Vol.3 で紹介した EAN は 13 桁の数字しか表現できませんでしたが、Code128 はアルファベットや記号も表現できる珍しいバーコードです。

1981 年にアメリカのコンピュータアイデンティクス社により開発された Code128 は、元々コンピュータへの入力用に ASCII コードに含まれる 128 種類の文字を表現できるものとして考案されました。これが Code128 の名前の由来にもなっています。

Code128 が表現できる文字には ASCII コードの制御文字も含まれますし、加えて 128~255 のバイト値も符号化できるようになっています[*3]。また規格上は桁数にも制限がないため、実質どんなバイト列でも表現できます[*4]。

現在では JIS X 0504 や ISO/IEC 15417 として規格化され、物流の分野を中心に世界中で使われています。

5.2　Code128 の構造

Code128 の構造を図 5.3[*5]に示します。

▲図 5.3　Code128 の構造

Code128 では、1~4 モジュール幅の黒か白のバー 6 本を合計 11 モジュール幅となるように並べたものを 1 シンボルキャラクタとして扱い、左から順に読んでいきます。

[*3] FNC4 シンボルと組み合わせることで表現しますが、詳細は割愛します。
[*4] 通常、Latin-1 の文字列として解釈します。
[*5] 画像出典：JIS X 0504

左端は表 5.1 に示す 3 種類のスタートキャラクタのいずれかで始まり、1 つ以上のデータキャラクタが並びます。そして 1 つのシンボルチェックキャラクタがあり、ストップキャラクタが右端になります。ストップキャラクタは 6 本の後ろに 2 モジュール幅の黒のバーを加えた 7 本で 13 モジュール幅です。また、両側には 10 モジュール幅以上の空白が必要です。

▼表 5.1　スタートキャラクタとストップキャラクタ

値	コード	エレメント幅	エレメントパターン
103	スタート A	2:1:1:4:1:2	
104	スタート B	2:1:1:2:1:4	
105	スタート C	2:1:1:2:3:2	
---	ストップ	2:3:3:1:1:1:2	

シンボルキャラクタ表

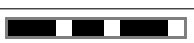
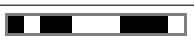
Code128 では、103 種類のシンボルが定義されており、シンボルキャラクタ値として 0~102 が割り当てられています。それぞれの値を文字に割り当てるコードセットは A・B・C の 3 種類があり、途中で切り替えることもできます。開始時のコードセットはスタートキャラクタによって指定されます。

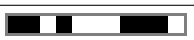
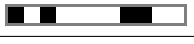
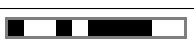
コード A は英大文字・数字・記号・制御文字、コード B は英大小文字・数字・記号が 1 キャラクタごとに割り当てられています。コードセット C は 1 キャラクタで数字 2 文字を表します。

ちょっと多いですが覚えてしまいましょう。バーの合計モジュール幅は偶数、スペースの合計は奇数になっているという特性を知っておくと覚えやすいです。

▼表 5.2　シンボルキャラクタ表

値	コード A	コード B	コード C	エレメント幅	エレメントパターン
0	SP	SP	00	2:1:2:2:2:2	
1	!	!	01	2:2:2:1:2:2	
2	"	"	02	2:2:2:2:2:1	
3	#	#	03	1:2:1:2:2:3	
4	$	$	04	1:2:1:3:2:2	
5	%	%	05	1:3:1:2:2:2	
6	&	&	06	1:2:2:2:1:3	
7	’	’	07	1:2:2:3:1:2	
8	(	(	08	1:3:2:2:1:2	
9	)	)	09	2:2:1:2:1:3	

値	コード A	コード B	コード C	エレメント幅	エレメントパターン
10	*	*	10	2:2:1:3:1:2	
11	+	+	11	2:3:1:2:1:2	
12	,	,	12	1:1:2:2:3:2	
13	-	-	13	1:2:2:1:3:2	
14	.	.	14	1:2:2:2:3:1	
15	/	/	15	1:1:3:2:2:2	
16	0	0	16	1:2:3:1:2:2	
17	1	1	17	1:2:3:2:2:1	
18	2	2	18	2:2:3:2:1:1	
19	3	3	19	2:2:1:1:3:2	
20	4	4	20	2:2:1:2:3:1	
21	5	5	21	2:1:3:2:1:2	
22	6	6	22	2:2:3:1:1:2	
23	7	7	23	3:1:2:1:3:1	
24	8	8	24	3:1:1:2:2:2	
25	9	9	25	3:2:1:1:2:2	
26	:	:	26	3:2:1:2:2:1	
27	;	;	27	3:1:2:2:1:2	
28	<	<	28	3:2:2:1:1:2	
29	=	=	29	3:2:2:2:1:1	
30	>	>	30	2:1:2:1:2:3	
31	?	?	31	2:1:2:3:2:1	
32	@	@	32	2:3:2:1:2:1	
33	A	A	33	1:1:1:3:2:3	
34	B	B	34	1:3:1:1:2:3	
35	C	C	35	1:3:1:3:2:1	
36	D	D	36	1:1:2:3:1:3	
37	E	E	37	1:3:2:1:1:3	
38	F	F	38	1:3:2:3:1:1	
39	G	G	39	2:1:1:3:1:3	
40	H	H	40	2:3:1:1:1:3	
41	I	I	41	2:3:1:3:1:1	
42	J	J	42	1:1:2:1:3:3	
43	K	K	43	1:1:2:3:3:1	
44	L	L	44	1:3:2:1:3:1	
45	M	M	45	1:1:3:1:2:3	
46	N	N	46	1:1:3:3:2:1	

値	コード A	コード B	コード C	エレメント幅	エレメントパターン
47	O	O	47	1:3:3:1:2:1	
48	P	P	48	3:1:3:1:2:1	
49	Q	Q	49	2:1:1:3:3:1	
50	R	R	50	2:3:1:1:3:1	
51	S	S	51	2:1:3:1:1:3	
52	T	T	52	2:1:3:3:1:1	
53	U	U	53	2:1:3:1:3:1	
54	V	V	54	3:1:1:1:2:3	
55	W	W	55	3:1:1:3:2:1	
56	X	X	56	3:3:1:1:2:1	
57	Y	Y	57	3:1:2:1:1:3	
58	Z	Z	58	3:1:2:3:1:1	
59	[	[	59	3:3:2:1:1:1	
60	\	\	60	3:1:4:1:1:1	
61	]	]	61	2:2:1:4:1:1	
62	^	^	62	4:3:1:1:1:1	
63	_	_	63	1:1:1:2:2:4	
64	NUL	`	64	1:1:1:4:2:2	
65	SOH	a	65	1:2:1:1:2:4	
66	STX	b	66	1:2:1:4:2:1	
67	ETX	c	67	1:4:1:1:2:2	
68	EOT	d	68	1:4:1:2:2:1	
69	ENQ	e	69	1:1:2:2:1:4	
70	ACK	f	70	1:1:2:4:1:2	
71	BEL	g	71	1:2:2:1:1:4	
72	BS	h	72	1:2:2:4:1:1	
73	HT	i	73	1:4:2:1:1:2	
74	LF	j	74	1:4:2:2:1:1	
75	VT	k	75	2:4:1:2:1:1	
76	FF	l	76	2:2:1:1:1:4	
77	CR	m	77	4:1:3:1:1:1	
78	SO	n	78	2:4:1:1:1:2	
79	SI	o	79	1:3:4:1:1:1	
80	DLE	p	80	1:1:1:2:4:2	
81	DC1	q	81	1:2:1:1:4:2	
82	DC2	r	82	1:2:1:2:4:1	
83	DC3	s	83	1:1:4:2:1:2	

値	コード A	コード B	コード C	エレメント幅	エレメントパターン
84	DC4	t	84	1:2:4:1:1:2	
85	NAK	u	85	1:2:4:2:1:1	
86	SYN	v	86	4:1:1:2:1:2	
87	ETB	w	87	4:2:1:1:1:2	
88	CAN	x	88	4:2:1:2:1:1	
89	EM	y	89	2:1:2:1:4:1	
90	SUB	z	90	2:1:4:1:2:1	
91	ESC	{	91	4:1:2:1:2:1	
92	FS	|	92	1:1:1:1:4:3	
93	GS	}	93	1:1:1:3:4:1	
94	RS	~	94	1:3:1:1:4:1	
95	US	DEL	95	1:1:4:1:1:3	
96	FNC3	FNC3	96	1:1:4:3:1:1	
97	FNC2	FNC2	97	4:1:1:1:1:3	
98	シフト	シフト	98	4:1:1:3:1:1	
99	コード C	コード C	99	1:1:3:1:4:1	
100	コード B	FNC4	コード B	1:1:4:1:3:1	
101	FNC4	コード A	コード A	3:1:1:1:4:1	
102	FNC1	FNC1	FNC1	4:1:1:1:3:1	

シンボルチェックキャラクタの計算

スタートキャラクタの値と、データキャラクタの値に重み（出現位置）を掛けた値をすべて合計し、それを 103 で割った余りがシンボルチェックキャラクタの値になります。

算出した値と読み取ったシンボルチェックキャラクタの値が一致するかどうかで、読み取りエラーかどうかを判定します。

このように、シンボルチェックキャラクタはデータではないため、バーコードリーダーでは通常表示しません。

5.3 演習

それでは、よく見かける箱に印字されていたバーコードを読んでみましょう。

▲図 5.4　印字されていたバーコード

1. 2:1:1:2:1:4 **スタート B**
2. 3:3:1:1:2:1 **56-B**「**X**」
3. 1:1:3:1:2:3 **45-B**「**M**」
4. 2:1:3:2:1:2 **21-B**「**5**」
5. 2:2:3:2:1:1 **18-B**「**2**」
6. 2:2:1:1:1:4 **シンボルチェックキャラクタ 76**
7. 2:3:3:1:1:1:2 **ストップ**

シンボルチェックキャラクタの値も計算して比較してみましょう。

$$104 + 1 \times 56 + 2 \times 45 + 3 \times 21 + 4 \times 18 = 385$$

$$385 mod 103 = 76$$

正しく読み取れたことがわかりました。

5.4　まとめ

この章では Code128 の基本的な読み方を解説しました。今回は特殊キャラクタ（シフト、FNC1〜4）の説明を省略しましたが、これは皆さんへの宿題とします。ぜひ JIS や ISO の規格書を調べてみてください。

これであなたもバーコードリーダー！

第 6 章

Python 組み込み関数マニアックス

Shunsuke Ito / @fgshun

6.1 はじめに

Python 組み込み関数。それはモジュールのインポートといった準備なしで用いることができる関数で、Python でプログラミングを始めると最初期から触れることとなるものです。しかしコードを読む機会が増えると、見慣れない使われ方をしているのを目にするようになってきます。そんな組み込み関数たちのいつもとは異なる姿を紹介してみます。

- print の動作は変えられる
- 数値型は引数なしで呼べる
- int は N 進数が読める
- next は StopIteration 例外を止められる
- sum は整数以外も足し合わせることができる
- max, min で比較する関数を指定できる
- iter とシーケンスプロトコル
- iter のもうひとつの機能 - 繰り返し呼び出す
- type のもうひとつの機能 - クラスオブジェクトを作る

6.2 print の動作は変えられる

`print`。Python に触れはじめたとき、まっさきに覚える組み込み関数はこれではないでしょうか。`print('Hello', 'World')`を実行したならば `Hello World`と画面に出力されるでしょう。では `print('Hello', 'World', sep=', ', end='!')`だとどうなるのでしょうか？ 答えは `Hello, World!`と出力されます。`print`関数にはいくつものキーワード引数があり動作を変更することができるのです。

まずは sep=’ ’（デフォルト値を=でつなげて表記します）。これは複数のオブジェクトを対象としたとき、それらの出力内容を区切る文字として使われます。

次に end=’\n’。出力の末尾につけられる文字です。

▼リスト 6.1　区切り文字列、末尾の文字列を変更する

```
>>> print(’Hello’, ’World’)
Hello World

>>> print(’Hello’, ’World’, sep=’, ’)
Hello, World

>>> print(’Hello’, ’World’, end=’!’)
Hello World!
```

file=sys.stdout。これは出力先を変更するための引数です。テキストファイルオブジェクトを指定してファイルに直接書いたりすることが可能です。それどころか write メソッドを持っているオブジェクトであれば出力先とすることができます。

つまるところ printは file引数のオブジェクトの writeメソッドをくり返し呼び出すという動きをするものなのです。リスト 6.2 はリストに文字列を保持する writeメソッドをもつクラス Xのインスタンスを file引数に渡してみたときの動作です。

▼リスト 6.2　write メソッド持ちのオブジェクトを出力先にする

```
>>> class X:
...     def __init__(self):
...         self.buffer = []
...     def write(self, string):
...         self.buffer.append(string)
...
>>> x = X()
>>> print(’Hello’, ’World’, file=x)
>>> x.buffer  # 4 つの文字列が収まっている。引数 2 つとその間に sep 最後に end
[’Hello’, ’ ’, ’World’, ’\n’]
```

最後は flush=False。Trueを与えると出力先がバッファを持っていて普段即時反映されない類のものであった場合でも強制的にフラッシュされるようになります。この動作は flushメソッドを呼び出すことでなされます。

▼リスト 6.3　flush メソッドが呼ばれていることの確認

```
>>> class Y:
...     def write(self, string):
...         pass
...     def flush(self):
...         print(’flush’)
...
>>> print(’Hello’, ’World’, file=Y(), flush=True)
flush
```

6.3 数値型は引数なしで呼べる

int,float,complex。Python でこれらを引数なしで呼んだことはあるでしょうか？空のコンテナを作るために set()などとしたことはあっても数値型に対して行ったことはないのではないでしょうか。これらの class たちは数値型であっても引数なしで呼ぶことが可能です。int()であれば 0、float()であれば 0.0、complex()であれば 0jが得られます。

この仕組みは、初期値を返す関数を引数にとるような設計のものと相性がよいです。たとえば collections.defaultdictにて初期値 0を設定するためには、lambda: 0ではなく intと書くことができます。

▼リスト 6.4　defaultdict と int を組み合わせて数え上げの仕組みを作る

```
>>> from collections import defaultdict
>>> counter = defaultdict(int)
>>> for v in ['spam', 'spam', 'spam']:
...     counter[v] += 1
...
>>> counter
defaultdict(<class 'int'>, {'spam': 3})
```

なお、引数なしで呼び出したときに得られる値は偽の値になっています。これは数値型に限らず、文字列、バイト列、コンテナ各種でも同様です。また、標準ライブラリが提供するクラス、たとえば collections.deque, decimal.Decimal, datetime.timedeltaなども同様の作りをしているものがあります。このことは値の変更もしくは値を収めた変数の内容の取り替えがなされたかどうかの判別に用いることができます。ただし、変更・取り替えの果てに偽の値に戻った場合をのぞきます。

▼リスト 6.5　組み込み型を引数なしで呼び出すと偽の値が得られる

```
>>> a = int()
>>> values = []
>>> for v in values:
...     assert v > 0
...     a += v
>>> if not a:
...     print('values is empty')
values is empty
```

6.4 int は N 進数が読める

Python の intは文字列を渡すと、これを intに変換します。この変換処理の前提にあるのは 10 進数が使われているということです。しかし、数値を表している文字列が 10 進数であるとは限らないでしょう。2 進数や 16 進数といった他の表現の方が読み書きしや

すい文脈は当然あります。そのため、10 進数以外の文字列を Python で読む機会もでてきます。

intには文字列を N 進数であるとみなして変換を試みるための base=10引数が存在します。たとえば int('11', base=2)とすると 3が、int('ff', base=16)とすると 255 が得られます。最大で 36 進数まで対応できます。 int('z', base=36)は 35です。

functools.partialと組み合わせると N 進数変換関数を作ることができます。リスト 6.6 は 9 進数変換関数を作るコードです。

▼リスト 6.6　partial と int を組み合わせて 9 進数変換関数を作る

```
>>> from functools import partial
>>> base9 = partial(int, base=9)
>>> base9('10')
9
```

base引数に 0 を渡したときは、Python コード中の整数リテラルとして解釈する動作に変わります。

▼リスト 6.7　base=0 は文字列を整数リテラルとして解釈する

```
>>> int('0xff', base=0)
255
>>> int('0b11111111', base=0)
255
```

6.5　next は StopIteration 例外を止められる

nextは Iteratorを引数にとり次の値を取得しますが、Iteratorが値を取り出し終わったときの StopIteration例外はそのまま再送出するという動きをします。というわけで、nextを使った処理をするときは try, except文を用いたコードになりがちです。ところで、この exceptが代わりの値に置き換えて終わり、といった簡素なものになったことはないでしょうか。この手の except文は next関数の第 2 引数 defaultを用いることで省くことが可能です。nextは第 2 引数が指定されたときに StopIteration例外が送られてくるとこれを握りつぶして第 2 引数を返します。ただし、第 2 引数を用いると Iteratorが終了したのか、終了していないがたまたま第 2 引数と同じ値が返ってきたのかの区別はつかなくなるので、その点には注意が必要です。

▼リスト 6.8　next の第 2 引数で StopIteration 例外を抑制

```
>>> it = iter([])
>>> next(it)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-bc1ab118995a> in <module>
----> 1 next(it)
```

```
StopIteration:
>>> next(it, 42)
42
```

6.6 sum は整数以外も足し合わせることができる

sumはひとつの Iterableから得られた値たちの合計を返すものです。ところで sum([])を実行するとどうなるのでしょうか？ 結果は 0となります。sumには合計を算出するときの初期値である第 2 引数 start=0[*1]が存在します。sumは値たちを start=0に対して +演算をくり返すという動作を行うのです。なにも加える対象がない場合にはただ start=0が返ります。

+演算子が使えるクラスのインスタンスであるのに sumの対象にすると TypeErrorになることがあります。これは 0との加算ができていないことが原因です。start=0引数に適当な初期値を指定することで回避することができます。リスト 6.9 は timedelta、時間オブジェクトの合計を求めるものです。

▼リスト 6.9　sum で時間の合計を求める

```
>>> from datetime import timedelta
>>> sum([timedelta(days=1), timedelta(days=2)])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-7ce5e0be73be> in <module>
----> 1 sum([timedelta(days=1), timedelta(days=2)])

TypeError: unsupported operand type(s) for +: 'int' and 'datetime.timedelta'
>>> sum([timedelta(days=1), timedelta(days=2)], timedelta())
datetime.timedelta(days=3)
```

ただし、文字列、バイト列は例外となっています。これは Python の str,byteには加算するたびに新しいオブジェクトを生成するという性質があるからです。大量の +演算での文字列連結は不得手とするところであり、sumは文字列、バイト列の連結を拒否します。エラーメッセージには素直に従いましょう。

▼リスト 6.10　sum を文字列に対して用いることはできない

```
>>> sum(['spam', 'ham', 'eggs'], start='')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-465b5510c047> in <module>
----> 1 sum(['spam', 'ham', 'eggs'], start='')

TypeError: sum() can't sum strings [use ''.join(seq) instead]
```

[*1] sum の第 2 引数は位置専用引数でした。Python 3.8 からはキーワード引数 start になり sum(a, start=b) と書くこともできるようになっています。

6.7 max, min で比較する関数を指定できる

max, minは引数たち、もしくはひとつの Iterableから得られた値たちの最大値、最小値を返すものです。これらには key引数が存在します。これを用いると値を直接比較するのではなく、指定された関数を用いて算出した値を比較するようになります。つまりどのような性質のものを大きい・小さいとみなすかを都度変更できるのです。

リスト 6.11 では、文字列の比較は辞書順で行われるところを長さで比較するよう変更しています。

▼リスト 6.11　min の key 引数に len を使用して長さ最小の文字列を探す

```
>>> # 通常は辞書順。なので c より a が優先される
>>> min(['aaa', 'bb', 'c'])
'aaa'
>>> min(['aaa', 'bb', 'c'], key=len)
'c'
```

6.8 iter とシーケンスプロトコル

iterは Iterableつまり__iter__メソッドをもつオブジェクトに対して用いて Iteratorを得るために使われます。ではそうではないオブジェクトを渡すとかならず TypeError例外になるのでしょうか？ いえ、シーケンスプロトコルに対応したオブジェクトに対してはその限りではありません。iterは__iter__メソッドがないオブジェクトを受け取ったとき、シーケンスプロトコルにしたがって値を取り出す Iteratorを作ることを試みます。

シーケンスプロトコルとは obj[0], obj[1], obj[2] …… といった 0 からはじまる連続した整数でのインデックスアクセスをくり返し行うものです。そしてシーケンスプロトコルに対応したオブジェクトは__getitem__メソッドを持っています[*2]。このため iterは渡されたオブジェクトに__iter__メソッドが見つからなかったときには、__getitem__メソッドがないかどうかを調べるのです。__getitem__も見つからなかった時はじめて TypeError例外となります。

▼リスト 6.12　シーケンスプロトコル対応インスタンスに対する iter

```
>>>  class A:
...      def __getitem__(self, key):
...          if 0 <= key < 5:
...              return key
...          raise IndexError
...
```

[*2] ___getitem___があるオブジェクトであっても必ずしもシーケンスオブジェクトに対応しているわけではありません。例としては dict があげられます。

```
>>> list(iter(A()))
[0, 1, 2, 3, 4]
```

6.9 iter のもうひとつの機能 - くり返し呼び出す

通常、 iterは引数をひとつ取り Iteratorオブジェクトを返す動作をします。では引数をふたつ渡すとどうなるのでしょうか？ ひとつめのオブジェクトをくり返し呼び出す Iteratorができあがるのです。この Iteratorはふたつめのオブジェクトに等しいオブジェクトが得られたときに停止します。使用例をあげます。

リスト 6.13 は iterでファイルオブジェクトの read1メソッドをくり返し呼び出す Iteratorを作り出し、これをもってバイナリファイルを読み出すコードです。iterを使わずにこの処理を書くのであれば、whileループを書くことになるでしょう。そんなループと同等の Iteratorを手軽に作ることができます。

▼リスト 6.13　iter, read1 でバイナリファイルを読み出す

```
>>> with open('binary_file', 'rb') as f:
...     while (data := f.read1()) != b'':
...         print('read {} bytes'.format(len(data)))
...
read 4096 bytes
read 3595 bytes

>>> with open('binary_file', 'rb') as f:
...     for data in iter(f.read1, b''):
...         print('read {} bytes'.format(len(data)))
...
read 4096 bytes
read 3595 bytes
```

6.10 type のもうひとつの機能 - クラスオブジェクトを作る

type。1 引数で用いるとそのクラスオブジェクトを取り出します。では 3 引数で用いるとどうなるでしょうか？ あらたなクラスオブジェクトができあがるのです。引数はそれぞれがクラス名、親クラスを並べた tuple、クラスの名前空間です。typeは class文と同等の機能を有します。リスト 6.14 における class文と typeの呼び出しは同じ結果が得られます。

▼リスト 6.14　type 関数で class を作る

```
>>> class X:
...     a = 1
...

>>> X = type('X', (object,), {'a': 1})
```

より正確には、class文による普段のクラス生成に使われる機構が type 3 引数なので、class文が typeと同等の機能となるのは必然です。そして typeもクラスオブジェクトであるので typeのサブクラスを作ることができます。このサブクラスを使うことでクラス生成処理をいつもの挙動から変えることが可能となります。class文で typeのサブクラスを使ってクラスオブジェクトを生成するようにするには metaclassキーワード引数を使います。

実は typeのサブクラスを 3 引数で呼んだ時に作り上げるのはクラスオブジェクトでなくてはならないといった制約は Python にはありません。リスト 6.15 は文字列を作ってしまう typeのサブクラスです。このようなものでも metaclass引数に渡すことができます。なので class文がつくるオブジェクトはクラスオブジェクトであるとは限らないのです。

▼リスト 6.15　文字列を作ってしまう type のサブクラス

```
>>> class StrMeta(type):
...     def __new__(cls, name, bases, classdict):
...         return name * 3
>>> class Spam(metaclass=StrMeta):
...     pass
>>> Spam
'SpamSpamSpam'
>>> type(Spam)
<class 'str'>
```

リスト 6.16 は enum.Enumの模倣をしてみるコードです。元のと比べると機能がまったく足りていませんが、それでも enum.Enumがどのように作られているのかを理解するためのとっかかりにはなります。

▼リスト 6.16　簡易 Enum

```
>>> class SimpleEnum(type):
...     def __new__(cls, name, bases, classdict):
...         clsobj = super().__new__(cls, name, bases, classdict)
...         for attr, val in classdict.items():
...             if attr.startswith('__'):
...                 continue
...             # クラス変数を置き換える後処理を入れる
...             attrobj = clsobj()
...             attrobj.name, attrobj.value = attr, val
...             setattr(clsobj, attr, attrobj)
...         return clsobj
>>> class X(metaclass=SimpleEnum):
...     a = 1
...     b = 2
>>> isinstance(X.a, X), X.a.name, X.a.value
(True, 'a', 1)
```

クラスの生成処理についての情報は公式ドキュメントの typeの項目とメタクラス[*3]の

[*3] https://docs.python.org/ja/3/reference/datamodel.html#metaclasses

項目、また PEP 3115[*4]に記載されています。

6.11 おわりに

ぜひ公式ドキュメントの組み込み関数[*5]の章を読み返してみてください。組み込み関数たちのまだ知らない使い方が見つかるかもしれません。ドキュメントでは物足りない、CPython でどのように実装されているのかを直接読みたいという方は `bltinmodule.c` [*6]に挑んでみるのもよいでしょう、ここに実装が集まっています。

[*4] `https://www.python.org/dev/peps/pep-3115/`
[*5] `https://docs.python.org/ja/3/library/functions.html`
[*6] `https://github.com/python/cpython/blob/master/Python/bltinmodule.c`

第 7 章

退屈なことは Emacs にやらせよう

Ryota Togai / @garicchi

7.1　イントロ

みなさまお好きなエディタは何でしょうか。最近だと Visual Studio Code や JetBrains 系という回答をされる方をよく見かけますが、ターミナルで動作する Vim や Emacs などの選択をされる方も多いかと思います。Vim や Emacs のよい点としては、エディタが対応しているスクリプトを自分で書けば、割となんでもできる自由度がある点です。自分は Emacs のキーバインドが好きなのと、Lisp という言語で設定を書ける点が好きなので Emacs を使っています。

Emacs には`--script`というオプションがあり、`--script`オプションに **Emacs Lisp**（以下 elisp）ファイルを渡すと、Emacs がそのスクリプトの中身を評価してくれます。さらに、elisp には`shell-command`という関数が存在し、引数にシェルコマンドを与えることで elisp からシェルコマンドを呼び出すことができます。この 2 つの機能を組み合わせると、まるで bash スクリプトのように Lisp でシェルコマンドのスクリプトを書くことができるようになります。

Lisp は書いていて気持ちがいい言語です。Lisp の構文は非常に単純で、基本的には'('と')'で囲まれる S 式しか存在しません。それゆえに人間の目には多少読みにくいプログラムにはなってしまいますが、この単純なカッコしかない文法の中に、自分のやりたいことを押し込めたときの快感はなかなかのものです。ぜひみなさんも Lisp を書いて、日常の作業を自動化しましょう。

7.2　Emacs Lisp 基礎

まずはみなさんおなじみの「hello world!」を出力してみましょう。リスト 7.1 に示すプログラムをエディタで書き、test.el として保存しましょう。

▼リスト 7.1　hello world!を表示する elisp

```
(message "hello world!")
```

そしてリスト 7.2 に示すコマンドを実行してみます。

▼リスト 7.2　実行する

```
$ emacs --script test.el
hello world!
```

「hello world!」と出力できました。

毎回 `emacs --script`とターミナルに書くのはめんどくさいので、シェルスクリプトのようにファイル名だけで実行できるようにしましょう。test.el をリスト 7.3 に示すコードに書き換えます。Emacs のパスはみなさんの環境に合わせて書き換えてください。

▼リスト 7.3　ファイル名だけで実行できるようにする

```
#!/usr/bin/emacs --script
(message "hello world!")
```

あとは通常のシェルスクリプトのように実行権限を与えて、相対パスで実行します。

▼リスト 7.4　実行する

```
$ chmod u+x test.el
$ ./test.el
hello world!
```

まるで bash スクリプトのように elisp を実行できるようになりました。このあと作っていくスクリプトはすべて上記の方法で実行していきます。

次に変数をつかってみましょう。リスト 7.5 は変数 xを 2 乗する elisp です。`let`はローカル変数 xを宣言し、10を代入。その後、後ろの式を評価します。`progn`は引数に与えられた S 式をすべて順番に評価し、最後の S 式の返り値を返します。`set`は変数へ代入を行います。今回は xを 2 乗したものを xに代入しています。Lisp は演算式をポーランド記法で書きます。美しいですね。

▼リスト 7.5　2 乗する elisp

```
#!/usr/bin/emacs --script

(message
 (let ((x 10))
   (progn
     (set 'x (* x x))
     (format "%d" x))))
```

▼リスト 7.6　2 乗する elisp 結果

```
$ ./test.el
100
```

リスト 7.7 は if 分岐をする elisp です。if関数は第 1 引数に真理値を返す S 式を書き、第 1 引数の真理値の結果から、trueの場合は第 2 引数を、falseの場合は第 3 引数を評価し、結果を返します。今回は x が 5 より大きいかどうかを条件式としましたが、条件式の場合は中置記法 (x > 5)のほうがわかりやすいかもしれません。でも美しいですね。

▼リスト 7.7　if 分岐する elisp

```
#!/usr/bin/emacs --script

(message
 (let ((x 10))
   (if (> x 5)
       "x > 5"
     "x <= 5")))
```

▼リスト 7.8　if 分岐する elisp 結果

```
$ ./test.el
x > 5
```

リスト 7.9 は 1〜10 までの数を総和する elisp です。総和を求めるときはまずループ制御構文で値を加算していくかと思います。elisp にも while ループがありますが、なるべく再帰を使うことでより美しくなります。

▼リスト 7.9　総和する elisp

```
#!/usr/bin/emacs --script

(message
 (format
  "%d"
  (let ((seq (number-sequence 1 10)))
    (progn
      (defun sum (seq)
        (if (null seq) 0 (+ (car seq) (sum (cdr seq)))))
      (sum seq)))))
```

まず、number-sequenceで 1〜10 の値を持ったリストを作ります。次に、sumという関数を定義します。sum関数は与えられたリストが nullなら 0を返し、nullでないならリストの carと再帰的に呼び出した sum関数の加算値を返します。carはリストの先頭部分を返す関数です。逆に cdrは先頭以外のリストを返します。このようにすることで、ca

rで徐々にリストから値を取り出し、再帰的に呼び出す関数には cdrで残りのリストを与えることで順番に値を処理していくことができます。

Lisp にはリストと、carと cdrというリストを処理する関数があるおかげで、再帰関数が書きやすくなっています。

▼リスト 7.10　総和する elisp

```
$ ./test.el
55
```

7.3　Emacs Lisp でシェルコマンドを呼び出す

elisp の基本知識がわかったところで、作業自動化に向けてシェルコマンドを呼び出してみましょう。

リスト 7.11 はルートディレクトリ直下で lib が名前に含まれるディレクトリを表示する elisp です。shell-command関数はシェルコマンドを呼び出す関数です。シェルを呼び出すのでパイプを利用することもできます。

▼リスト 7.11　シェルコマンドを呼び出す elisp

```
#!/usr/bin/emacs --script

(shell-command "ls / | grep lib")
```

▼リスト 7.12　シェルコマンドを呼び出す elisp 結果

```
$ ./test.el
lib
lib64
```

しかし上記の elisp では、コマンドの出力結果が標準出力に出てしまいました。これでは出力結果をつなぎ合わせたりするのに不便なので、コマンド出力結果もリストで取得しましょう。

リスト 7.13 はシェルコマンドを呼び出して結果をリストとして取得する elisp です。shell-command-to-stringはシェルコマンドを呼び出し、標準出力を文字列として取得する関数です。split-stringは文字列を第 2 引数に指定したデリミネーターで分割し、リストとして返す関数です。prin1はリストを表示する関数です。

▼リスト 7.13　シェルコマンドの結果を list として取得する elisp

```
#!/usr/bin/emacs --script

(prin1
 (split-string
  (shell-command-to-string "ls / | grep lib")
  "\n"))
```

これでシェルコマンドの呼び出し結果をリストとして取得できました。

▼リスト 7.14　シェルコマンドの結果を list として取得する elisp

```
$ ./test.el
("lib" "lib64" "")
```

リストにさえしてしまえば、Lisp の世界ではとても扱いやすくなります。上記の、コマンドを実行して結果をリストで取得することを繰り返せば、Lisp で書けるシェルスクリプトの完成です。

7.4　Emacs Lisp で作業を自動化する

それでは最後に、今まで学んだ elisp スクリプトで、作業を自動化してみましょう。

退屈な作業パターンとして、文章を大量生成する必要があるパターンを考えます。たくさんの人にメールを送ったり、毎回同じような定型文を作る必要がある場合、毎回手打ちするのは非常に退屈な作業です。そこで文章のテンプレートを用意してそれを必要な分だけ置換して文章を生成しましょう。

まずはリスト 7.15 に示すテンプレートを考えます。リスト 7.15 を template1.txt として保存しましょう。

▼リスト 7.15　template1.txt

```
こんにちは{{NAME}}さん。
今日はいかがお過ごしでしょうか。
{{DATE}}の予定はいかがですか。
お返事お待ちしております。
```

{{NAME}}や{{DATE}}のように置換したいところをわかりやすい文字列で書きます。テンプレートファイルを読み込んで必要な部分を置換する elisp を書けば文章を大量生成できそうです。

リスト 7.16 がテンプレートから文章を生成する elisp です。

▼リスト 7.16　テンプレートを使って文章を生成する elisp

```
#!/usr/bin/emacs --script

(let
    ((tempfile (elt argv 0)) ; 引数からテンプレートファイルパスを取得
     (people '(("田中" "12/1") ("佐藤" "1/5")))) ; 置換するデータ
```

```
(defun replace(people)
  (if (null people) ; people が空になるまで再帰で呼び出される
      nil
    (progn
      (let ((name (caar people))
            (day (cadar people))
            (temp (shell-command-to-string (format "cat %s" tempfile)))) ; cat で読み込む
        ;; replace
        (set 'temp (replace-regexp-in-string "{{NAME}}" name temp))
        (set 'temp (replace-regexp-in-string "{{DATE}}" day temp))
        (shell-command-to-string (format "echo \"%s\" > %s.txt" temp name)) ; 書き込み
        (replace (cdr people))
        ))))
(replace people))
```

まず、コマンドライン引数でテンプレートファイルのパスを取得します。elisp ではコマンドライン引数は argvという変数に保存されます。そこから eltt関数を使って 0 番目をとりだします。

people変数には置換したいデータを格納し、replaceという関数を定義、再帰的に呼び出して人数分処理します。

caarと cadarは (car (car LIST))と ((car (cdr (car LIST))))と同義です。リストの必要な部分まで瞬時にアクセスできます。

(shell-command-to-string (format "cat %s" temfile))は catコマンドを呼び出し、ファイルの中身を stringとして取得します。

あとは replace-regexp-in-stringで文字列置換を行い、echoとリダイレクトでファイルに保存します。

実行すると、リスト 7.17 のようにテンプレートを読み込んで文章を生成してくれます。

▼リスト 7.17　文章を生成した結果

```
$ ./test.el template1.txt

$ ls
template1.txt  test.el  佐藤.txt  田中.txt

$ cat 佐藤.txt
こんにちは佐藤さん。
今日はいかがお過ごしでしょうか。
1/5 の予定はいかがですか。
お返事お待ちしております。
```

7.5　おわりに

Q. bash スクリプトでよいのでは？

A. それを言ってはいけない

第 8 章

CloudFront-WAF 制御下で Lambda@Edge を利用して ReactSPA を動かす

Daisuke Yamaguchi / @__gutio__

React[*1]などの SPA フレームワークで構築したサイトを、各種のマネージドサービスを利用して公開する機会が増えてきました。ただし一般公開前のステージング環境については、社内 IP などに制限して公開することが必要になると思います。

この章では、AWS[*2]上で制限公開時に発生する問題を Lambda@Edge[*3] というサービスを用いて解決する方法をご紹介します。

8.1 S3 と CloudFront を利用してウェブサイトを公開する

まずは、AWS でマネージドサービスを利用して公開する方法を簡単に紹介します。

AWS で静的ソースのみでサイトを公開する際には、一般的に S3[*4]というオブジェクトストレージサービスを利用して公開します。S3 の機能に静的ウェブサイトホスティング[*5]のオプションがあるので、S3 のみで公開することも可能ですが、それでは http での配信しかできません。そのため最近の常時 https 通信を求める環境においては、CloudFront[*6]という CDN サービスを S3 の前段に利用して https での通信を可能にします。

[*1] https://ja.reactjs.org/
[*2] https://aws.amazon.com/jp/
[*3] https://aws.amazon.com/jp/lambda/edge/
[*4] https://aws.amazon.com/jp/s3/
[*5] https://docs.aws.amazon.com/ja_jp/AmazonS3/latest/dev/WebsiteHosting.html
[*6] https://aws.amazon.com/jp/cloudfront/

1. S3 バケットへファイルを配置する

web サイトで配信する静的ファイルを格納した S3 バケットを用意します。
この際、注意点がいくつかあります。

- CloudFront から S3 へアクセスするため、静的ウェブサイトホスティングのオプションの有効化は不要です。
- サイトのファイル自体は静的ウェブサイトホスティングするときと同様にバケット直下に配置してください。

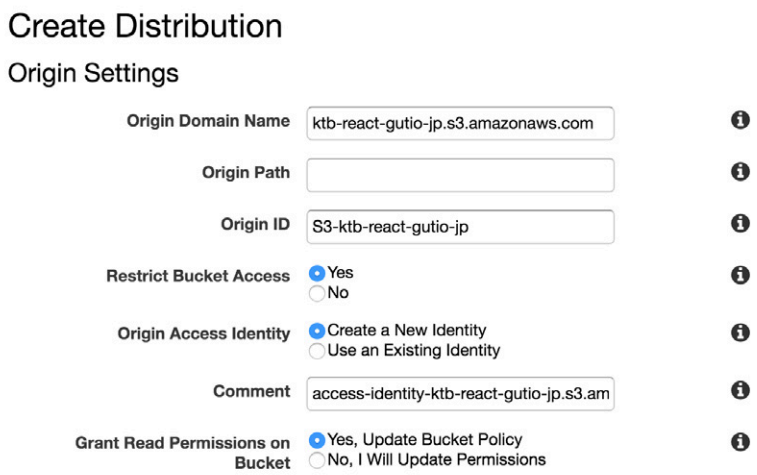
今回、React での記述法などはメインターゲットではないため、公開されていたデモ[*7]をお借りしました。
path 設定などを書き換えてビルドし、S3 のバケットへ配置すると以下の画像のようになります。

▲図 8.1　S3 バケットへのファイル配置例

[*7] https://codesandbox.io/s/nn8x24vm60 BootStrap のリンク切れなどは一部修正して利用しています

2. CloudFront のディストリビューションを作成する

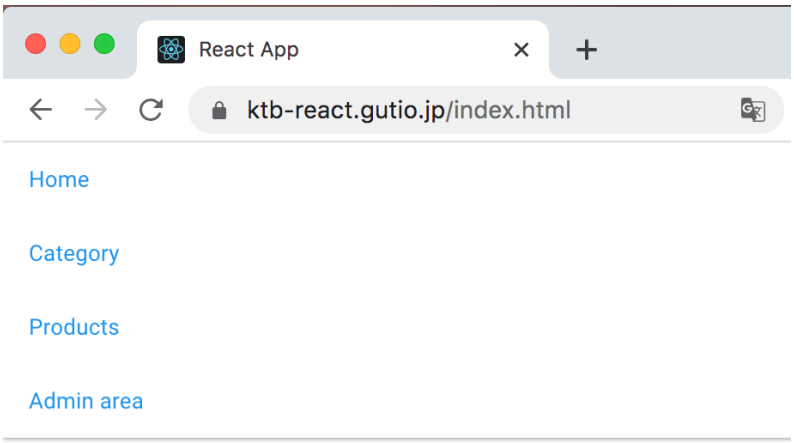
CloudFront でコンテンツ配信をするための設定することを「ディストリビューションを作成する」といいます。ディストリビューションの作成ウィザードで、用意した S3 バケットの内容を公開するために次のように設定します。



▲図 8.2　ディストリビューションの作成

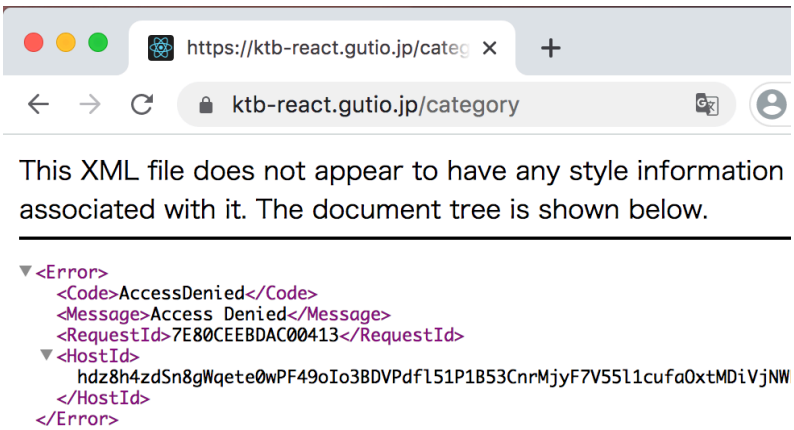
- Origin Domain Name ： S3 バケットをセットします。
- Restrict Bucket Access： ON にしてください。private な S3 バケットへ次の「Origin Access Identity」という AWS 内部での権限設定を利用してアクセスします。
- Origin Access Identity：S3 へアクセスする時に利用する権限を選択します。今回はディストリビューション作成時に一緒に新規に作成させるため「Create New Identity」を選択しています。
- Grant Read Permissions on Bucket：Yes を選択します。ディストリビューション作成時に、S3 側のアクセス設定にこのディストリビューションの「Origin Access Identity」設定を受け入れるよう変更します。

その他のディストリビューション設定はデフォルトのまま作成します。
アクセスするとこのようになります。



▲図 8.3　index.html へのアクセス

しかし、このままだと最初にページを開く際に `/index.html` の path でアクセスしないとこのような 403 エラーになってしまいます。

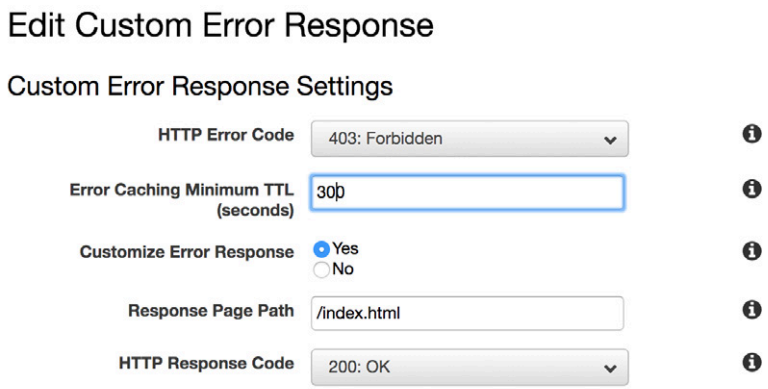


▲図 8.4　/category への直接アクセス (403 Error)

3. カスタムエラーレスポンスを設定する

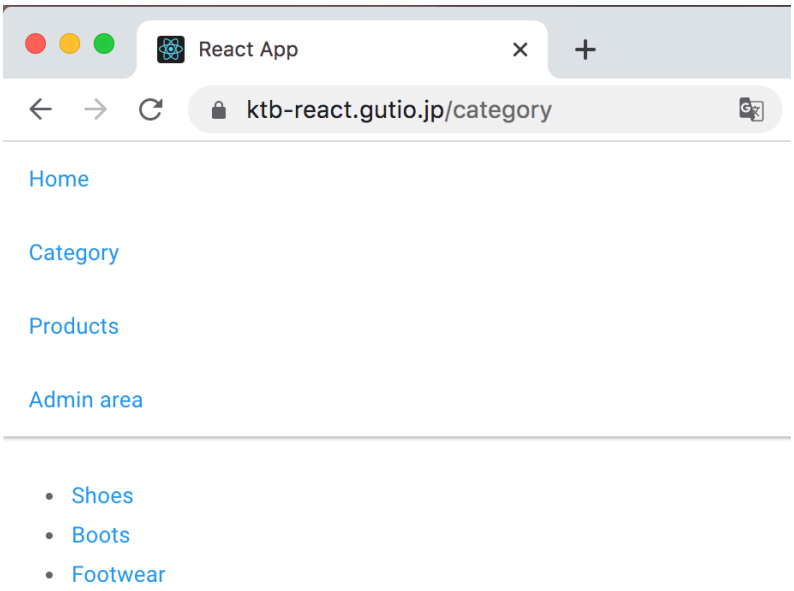
React の SPA ではコンテンツを切り替える際に React Router というルーティングコンポーネントを利用します。`/` や `/path` などの React Router で設定されるパスへ直接アクセスしても実態のファイルがない状態でも適切に表示するには、カスタムエラーレス

ポンスを設定する必用があります。今回の構成で実態のファイルがない場合には 403 エラーになるので、以下のようにレスポンスコードを 200 に書き換えて `/index.html` のコンテンツを返すよう設定します。



▲図 8.5　カスタムエラーレスポンスの設定

このように設定することでブラウザリロードしてもエラーにならずに表示できます。



▲図 8.6　/category への直接アクセス (Success)

8.2 CloudFront に WAF で IP 制限を適用する

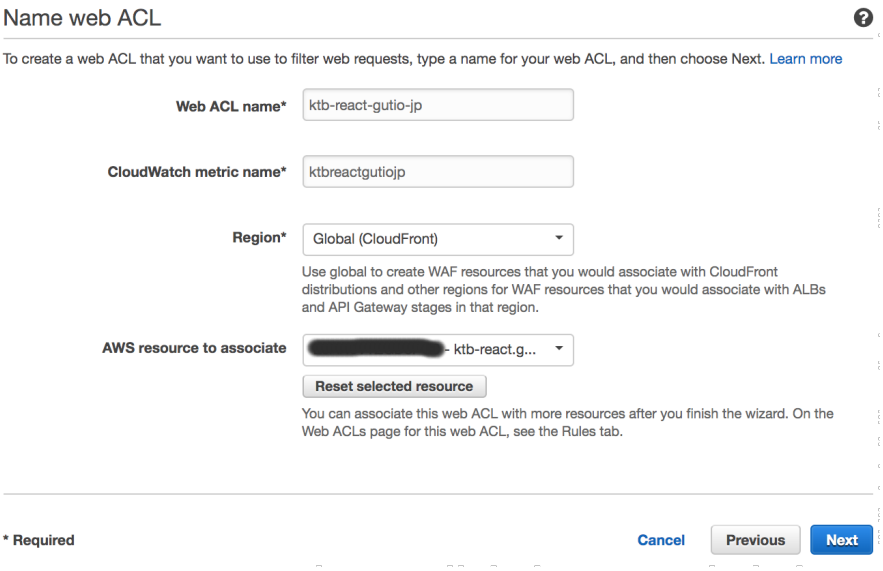
AWS WAF (Web Application Firewall)[*8] とは、CloudFront などに設定ができるファイアーウォールツールです。CloudFront で構築したステージング環境を社内 IP からのアクセスのみに制限するためにこれを利用します。

WAF でのアクセス制御には IP 以外にもヘッダ内の文字列マッチなども可能です。今回はこちらを利用して、ブラウザが Chrome だった場合のみアクセス許可する方法を説明します。

つい先日 WAF に新しいバージョンがリリースされましたが、今回は旧バージョンの Classic を利用する方法を説明します。

Step 1. New web ACL

WAF での設定を作成するには ACL[*9] というものを作成します。任意の ACL の名前と設定の対象となる CloudFront を選択します。



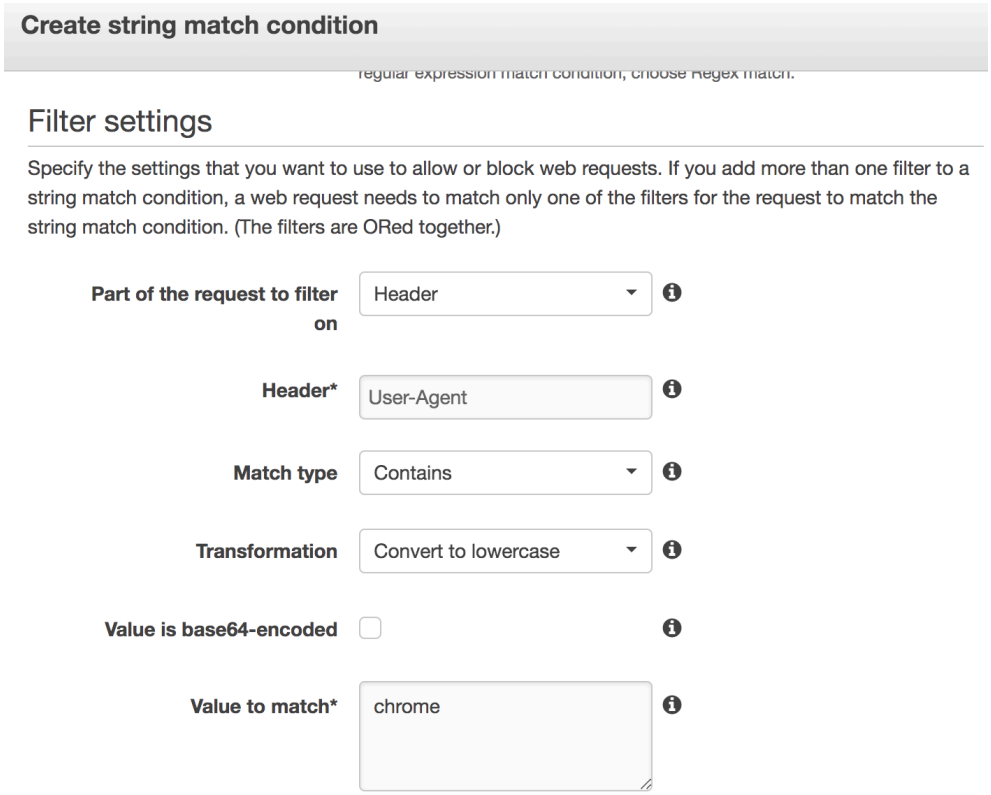
▲図 8.7　ACL の作成 step1

[*8] https://aws.amazon.com/jp/waf/

[*9] https://docs.aws.amazon.com/ja_jp/waf/latest/developerguide/web-acl.html

Step 2. Create Conditions

今回はアクセスブラウザが Chrome だった場合に制限するため、UserAgent をすべて小文字に変換してに「chrome」という文字列が含まれているかどうかを判定する設定を作成します。そのために設定する項目は「String and regex match conditions」になります。



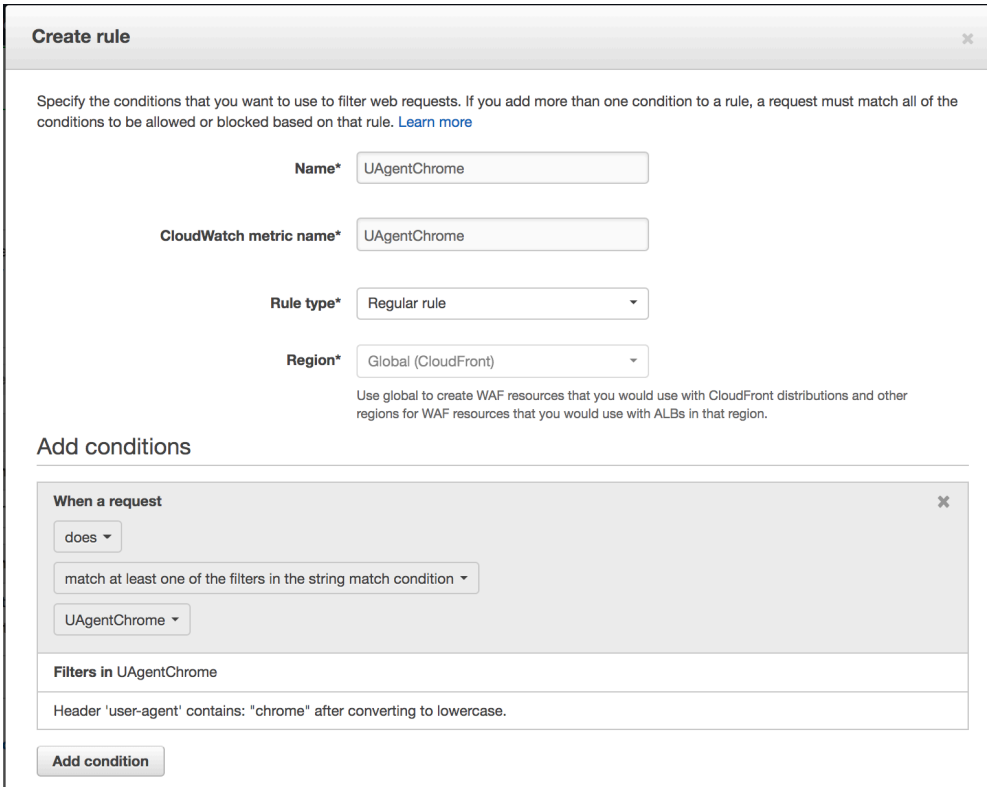
▲図 8.8　ACL の作成 step2 (string match の作成)

社内 IP のみに制限する場合には、「IP match conditions」を利用して対象 IP に含まれるかどうかを判定するものを作成してください。

Step 3. Create Rules

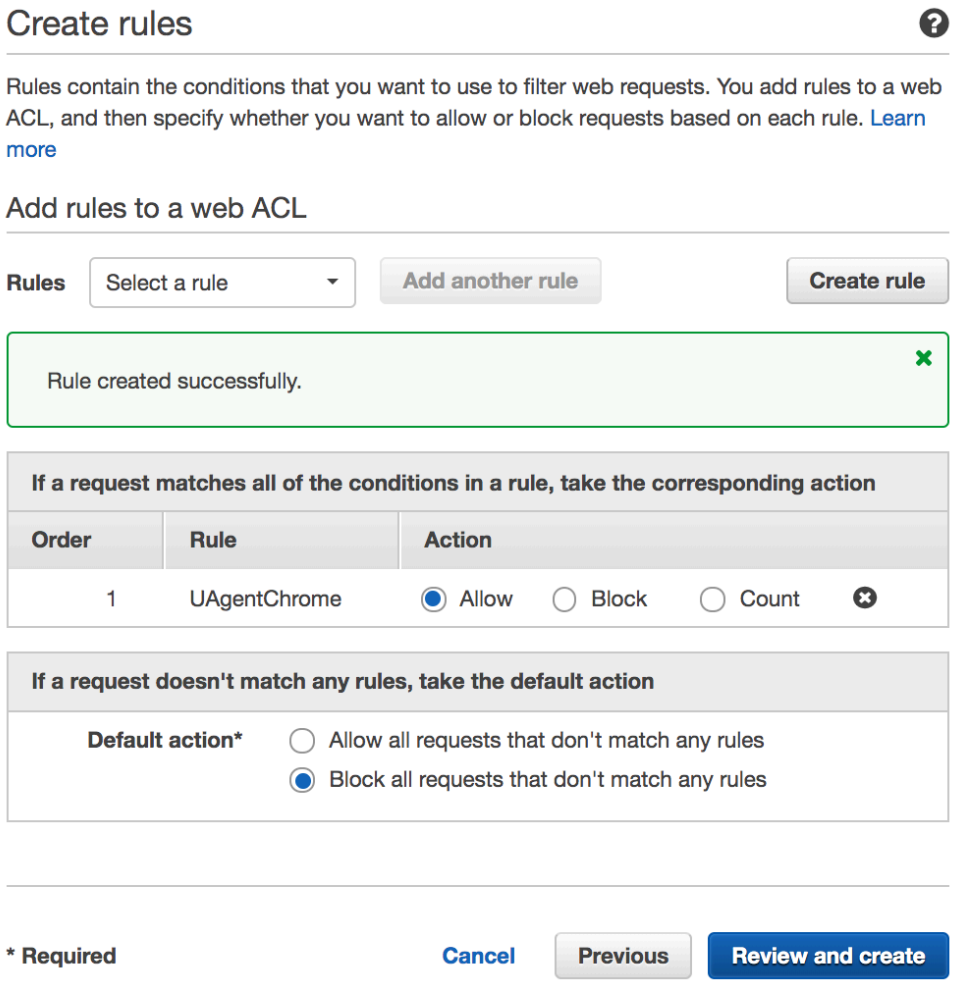
Step 2 で作成した設定をもとに振り分けルールを作成します。

- RuleType：「Regular Rule」を選択します。任意の名前を設定してください。
- condition：作成した「string match condition」を設定します。



▲図 8.9　ACL の作成 step3 (Rule の作成)

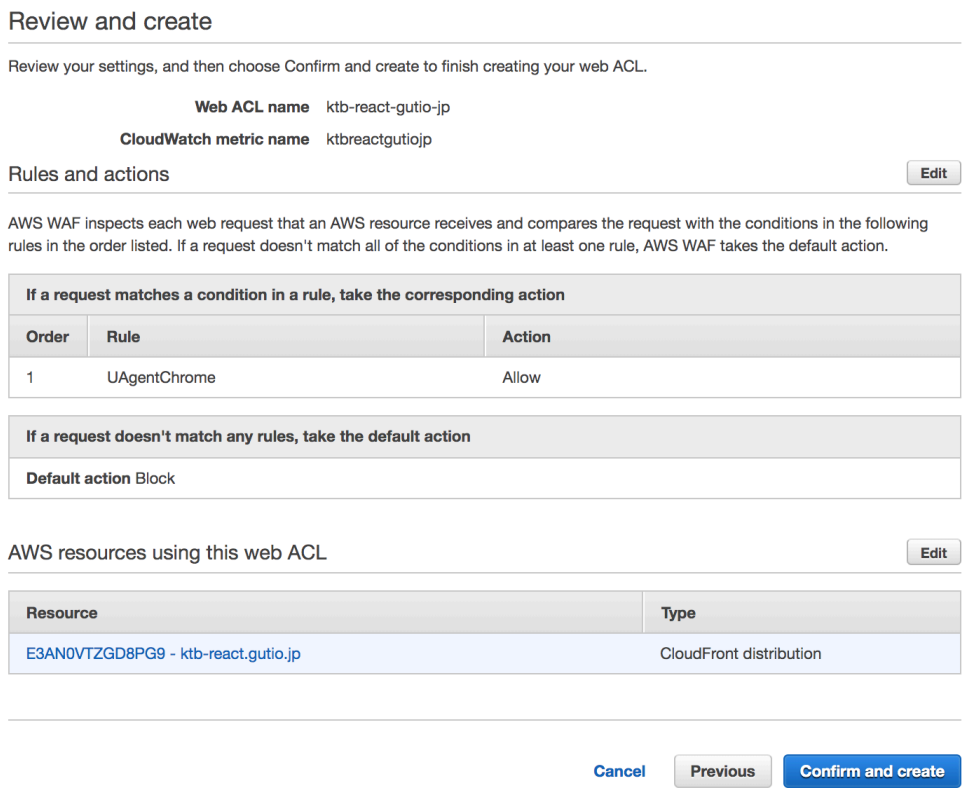
ルールは上から順に判定していき、マッチするパターンに応じて許可不許可を判定します。すべてにマッチしなかった場合には最後の「Default」設定が適用されます。



▲図 8.10　ACL の作成 step3

Step 4. Review and create

最後に想定どおり設定が入っているかを再度確認して作成します。



▲図 8.11　ACL の作成 step4

8.3　403 エラー

Safari などのブラウザでアクセスして WAF によりアクセス遮断された場合、本来はこのような 403 エラーをレスポンスします。

▲図 8.12　WAF によるアクセスエラー

しかし、この章の最初から順に設定していると Chrome 以外のブラウザでは何も表示されません。そこで curl などを利用して UserAgent を指定せずに URL へアクセスして中身を見てみましょう。`/index.html` の中身が返ってきていることがわかります。

これは CloudFront で、 WAF での 403 エラーと S3 アクセス時に実体ファイルがないときの 403 エラーの区別がつかないため、どちらもまとめて 403 のカスタムエラーレスポンスの設定にしたがってレスポンスをしているためです。ただしその他の css や javascript のファイルへのアクセスも 403 になって `/index.html` が返ってきているため、ページ全体として今回は真っ白な表示になりました。ですが `/index.html` に含まれる情報であるページのタイトルなどは確認できてしまっています。

つまりこのままでは `/index.html` 自体に記載されたコンテンツは、ステージング環境の URL で一般公開されているのと同じになってしまいます。

8.4　Lambda@Edge を利用して対応する

403 エラー が共通してしまいステージング環境での意図しない一般公開を避けるためには、カスタムエラーレスポンスの設定を解除しなければなりません。しかしそうなると、

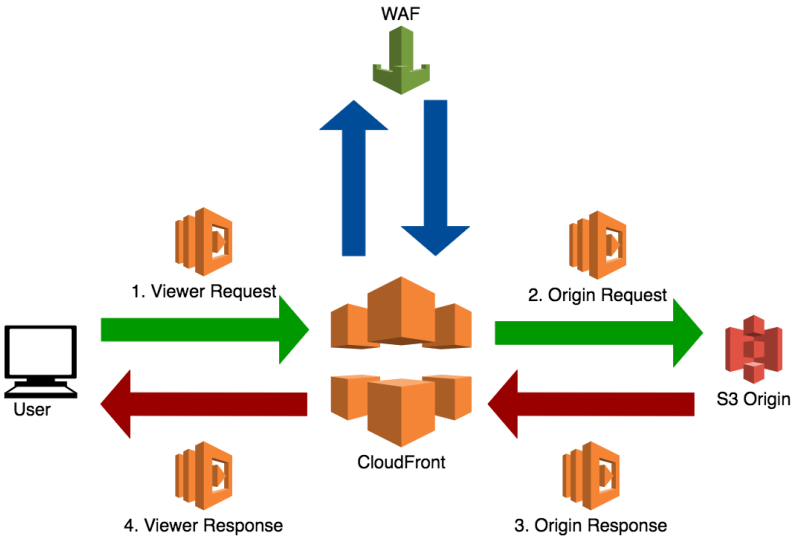
ユーザーがブラウザでリロードした際などにページが表示されなくなってしまいます。

この問題の解決が、Lambda@Edge を使うことで可能です。

Lambda@Edge とは？

AWS Lambda とは、AWS で利用可能な FaaS（Function as a Service）ですが、その中でも Lambda@Edge とは、CloudFront のエッジロケーションで実行されるものを指します。

Lambda@Edge の実行トリガには 4 種類の実行タイミングから選択できます。



▲図 8.13　CloudFront と Lambda@Edge と WAF の関係イメージ

1. Viewer Request：クライアントから CloudFront へのリクエストが来た際に実行されます。
2. Orijin Request：CloudFront から S3 などの Origin へリクエストする前に実行されます。
3. Origin Responce：S3 などの Origin からのレスポンスが帰ってきた際に実行されます。
4. Viewer Responce：クライアントへのレスポンス前に実行することができます。

8.5 ビューアーリクエストの際にアクセスリソースのパスを書き換える

この Lambda@Edge を利用することで、特定の path にアクセスが来た場合のみ内部でのアクセス先を `/index.html` に書き換えるよう制御が可能です。これで S3 アクセスの際に 403 エラーを発生させなくできるので、カスタムエラーレスポンスの利用が不要になります。

1. Lambda@Edge 関数の作成

AWS Lambda には nodejs, Python, Ruby, Java, Go, .Net Core というさまざまなランタイムが提供されています。しかし、Lambda@Edge の関数を作成する際に気をつけなければならない点が２つあります。

まずは、利用可能なランタイムはその中でも次の２種類に限られている点です。

- nodejs 10.x （8.10 も実行されていますが、新規での作成は停止されています）
- Python 3.7

nodejs 12.x や Python 3.8 も Lambda としては指定可能ですが、使えませんのでご注意ください。

次の注意点は Lambda@Edge 関数を作成するリージョンです。国内で開発していると開発環境のリージョンには東京や価格設定の安いバージニア・オレゴンあたりを設定することが多いかと思います。ですが Lambda@Edge に指定できる関数の作成リージョンは、us-east-1 （バージニア北部）に限られますので気をつけてください。

これらの注意点を考慮して、今回は nodejs 10.x のランタイム向け関数を作成します。

▲図 8.14　Lambda 関数の作成

Lambda@Edge もまた、CloudFront 同様ログ出力などのためにアクセス権限の設定が必用です。関数作成時のオプションで Lambda@Edge 向けのポリシーテンプレートを使ってアクセス権限を設定するロールを作成してください。

作成する Lambda 関数のコード[*10]は次のとおりです。

[*10] https://gist.github.com/gutio/b3709f35cd65df7bae0104217503bc4c

▼リスト 8.1　特定 path を /index.html に差し替える Lambda@edge スクリプト

```
// ReactRouter で設定している特定 path を /index.html に差し替える Lambda@edge スクリプト
exports.handler = (event, context, callback) => {
    var request = event.Records[0].cf.request;

    // ReactRouter で設定している Path の列挙
    var spaPages = [
        '/',
        '/category',
        '/category/shoes',
        '/category/boots',
        '/category/footwear',
        '/products',
        '/products/1',
        '/products/2',
        '/products/3',
        '/products/4',
        '/login',
        '/admin'
    ];

    // リストに含まれるパスへのアクセスだった場合
    if(spaPages.indexOf(request.uri) >= 0) {
        // /index.html へのアクセスに差し替える
        request.uri = '/index.html';
    }
    return callback(null, request);
};
```

2. Lambda@Edge 関数として CloudFront へ設定する

作成した Lambda 関数を Lambda@Edge 関数として CloudFront に設定することは関数画面上部のアクション項目から可能です。

▲図 8.15　Lambda@Edge として設定 1

対象となるディストリビューションを選択し、関数を実行する CloudFront イベントとしては ビューアーリクエストを選択します。

▲図 8.16　Lambda@Edge として設定 2

これで設定完了です。Chrome と Safari などブラウザを使い分けたり遷移後の URL でリロードして確認してみてください。

その他のアプローチ

今回はビューアーリクエストで発火する Lambda@Edge のスクリプト中に、事前にパスをリストアップしておいて該当する場合は `/index.html` へのアクセスに繋ぎ変える方法を紹介しました。ですが、他にもいろいろなアプローチが考えられます。

１つは、Lambda@Edge から参照可能なリソースに DynamoDB があるので、スクリプト中に書いた対象 PATH のリスト管理に利用する方法です。デプロイ時に合わせてリストを更新することで、対象 PATH の管理をスクリプト自体とは別で設定する方法が考えられます。

また、特定パスへのリクエストをルーティングするという点では CloudFront のビヘイビアでの特定パスパターンへ設定することも可能だと思います。SPA で設定するユー

ザー向けパスについて特定のパスパターンのルールを作ることで、ビヘイビアでマッチした場合に実行される Lambda@Edge 自体は無条件に `/index.html` へルーティングを変えるものにできそうです。

ほかには、Lambda@Edge をオリジンレスポンスに紐付けること方法です。オリジンアクセスで 403 になった場合のみ改めて `/index.html` の中身を取得して、その中身を 200 レスポンスとして返却する方法も考えられます。

8.6 まとめ

WAF による制限適用下での ReactRouter による SPA を、ブラウザリロードなどの直接アクセスに対応させる方法を紹介しました。

紹介した方法は簡単な一例でしかありません。ぜひご自身の環境に合わせたアプローチを選択して快適な開発をおすごしください。

8.7 デモサイト

このデモでは記事で紹介したように、WAF でのアクセス制御に IP 制限ではなく User-Agent を小文字変換した際に chrome という文字列が含まれるかどうかで制御しています。Chrome と Safari などブラウザを使い分けてぜひアクセスしてみてください。

今回のデモサイト： `https://ktb-react.gutio.jp/`

第 9 章

天下一 Game Battle Contest（β）の裏側

Takuya Hashimoto / @hasi_t

天下一 Game Battle Contest（GBC）[*1]は、KLab が開催するプログラミングコンテストです。2019 年 12 月 21 日にオープンベータテストを開催しました。2020 年には本番のコンテストを開催予定です。

KLab は 2012 年から毎年 AtCoder 上でプログラミングコンテスト「天下一プログラマーコンテスト」を開催していて、2019 年はそれに追加して GBC も開催しました。当初は 2017 年度に開催予定だったのですが、いろいろあって 2019 年になってしまいました。

天下一プログラマーコンテストと GBC は、「競技プログラミング」のコンテストです。AtCoder 上の天下一プログラマーコンテストは、基本的には正解・不正解が明確な「アルゴリズム」のコンテストであるのに対し、GBC には正解・不正解という概念はなく、得点を競う「マラソン」と呼ばれる種のコンテストです。最近は短時間のマラソンも増えましたが、基本的には数日から数週間と期間が長めのコンテストが多いです。GBC は 4 時間のコンテストでした。これは短い部類です。

また、GBC は参加者側でプログラムを実行する形式のコンテストです。運営側は API サーバを提供し、参加者はサーバと HTTPS で通信するプログラムを作成・実行し、API サーバ側で得点を計算します。

GBC と同じ形式のコンテストは、CodeRunner[*2]というコンテストが 2014 年と 2015 年に開催されていました。競技者として参加していてとても楽しいコンテストでしたし、ゲーム API サーバを作るというのは本業なので、自分も開催したいという気持ちを刺激されるコンテストでした。CodeRunner が開催されなくなってしまったので、これを好機とみて 2017 年には開催に向けて動き始めました。しかし、ISUCON7 の出題をしたり、本業が多忙になったり、作問メンバーが転職したりで、2019 年になってしまいました。

[*1] https://tenka1.klab.jp/2019-obt/

[*2] https://twitter.com/coderunner_jp

2019 年は、コンテスト開催に必要なものの洗い出しと、この形式のコンテストを開くという意志を見せることを目的としてオープンベータテストを開くことにしました。

9.1 オープンベータテストの問題概要

- 20 × 20 頂点のグリッドグラフがあり、ゲームごとに各辺の初期容量と source, sink の頂点集合が与えられる。
- 参加者は API サーバへの通信によって辺を選択し、選択した辺について（初期容量/選択した人数）の容量の辺を張り、そのグラフの最大流が得点となる。
- 1 分 1 ゲームで、240 ゲームを行う。

詳細は `https://tenka1.klab.jp/2019-obt/` を参照ください。

9.2 作問にあたって考えたこと

まず、ゲームルールの単純さです。時間が短いので、複雑なゲームは適しません。今回の問題は辺を選ぶだけの単純なゲームなので、この点は問題ないと思っています。

ちなみに、辺取りゲームという題材は 2017 年の ICFP Programming Contest の影響が大きいです。この大会では KLab の社員・元社員を中心としたチーム DiamondPrincess で準優勝したので、個人的にも思い出深いコンテストです。ICFPC のほうはターン制で他の人が取った辺は取れなくなるのに対して、GBC ではリアルタイム制で他の人の影響は受けずに辺は取れる、ただし容量が減る、というルールにしました。

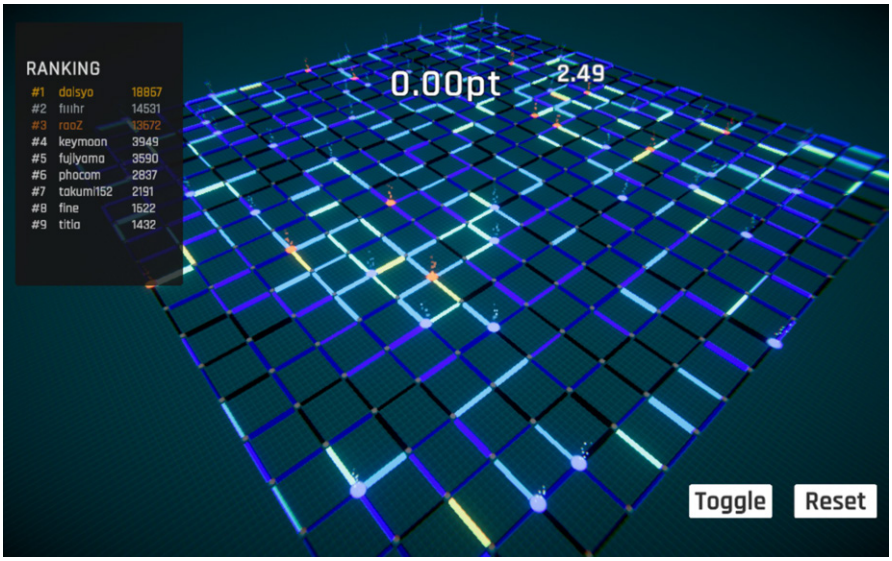
単純であっても簡単に最適解が求まるようではマラソン形式のコンテストとしては不適切です。競技時間中スコアを伸ばす余地が有り続けるのが理想です。最大流を使ってスコアを決めるというのと、他の人の影響で容量が減るルールは、最適解を求められなくする意図があります。

また、プログラミングコンテストなので、手動である程度遊べるのはよいのですが、最終的にはプログラムを使わないと勝てないようにする必要があります。最初に作問チームでテストプレイしたときは 1 秒ごとに 1 辺までに制限にしていたら手動で遊び続けた人が勝ってしまったので、0.5 秒ごとに 1 辺までに制限にすることでプログラムが有利になるようにしました。

9.3 ビジュアライザ

GBC では視覚的にゲームの内容が理解でき、手動操作で参加可能なビジュアライザを用意しました。これは、参加者に見た目でもゲームを楽しんでもらい、また競技に取り組みやすくすることを目的としています。Web ブラウザで遊べるように、Unity で作成して WebGL ビルドしたものを提供しました。ゲーム会社らしさを見せたいという気持ちもあったので、その目的も達成できたのではと思っています。動いているビジュアライザ

の様子は https://twitter.com/klab_tenka1/status/1208294269481472000 にあります。



▲図 9.1　ビジュアライザ画面

9.4　API サーバ

コンテストのゲーム API サーバ作りは、業務と違ってコンテストが終わった後の運用を気にしなくて良い、というのが業務と違うところです。これは業務で使いづらい技術を使う良い機会です。今回は OpenResty と Redis で API サーバを作成しました。ひとつの nginx.conf ファイルだけで完結していて、nginx 上で動く Lua スクリプトと Redis 上で動く Lua スクリプトがさも当然のように混ざっているのがお気に入りです。スコア計算は別途バッチサーバで実行する作りにしました。

API サーバのソースコードは https://github.com/KLab/tenka1-2019-obt で公開しています。

9.5　おわりに

天下一 Game Battle Contest の本番は、2020 年夏開催予定です。楽しいコンテストになるよう鋭意準備中です。本番の情報は天下一プログラマーコンテスト公式サイト https://tenka1.klab.jp/ と公式 Twitter アカウント https://twitter.com/klab_tenka1 で発信予定です。参加をお待ちしています！

著者

第 1 章 Sho HOSODA / @gam0022
シェーダー芸人

第 2 章 Shinya Naganuma / @Pctg_x8
Unity 半年ぶり (リアル) に触った

第 3 章 Kinuko Mizusawa
みんな好きですが糸繰叶ちゃん推しです

第 4 章 Toshifumi Umezawa
メイン PC をタブレットに移行したら、マイコン繋ぐ USB 端子足りない問題と電力不足問題が発生

第 5 章 Daisuke Makiuchi / @makki_d
眼鏡っ娘が好きです

第 6 章 Shunsuke Ito / @fgshun
初心に返って Python ドキュメント読み。組み込み関数

第 7 章 Ryota Togai / @garicchi
親からの結婚プレッシャーをどう回避するかが最近の悩み

第 8 章 Daisuke Yamaguchi / @_gutio_
GCP も触らなきゃならないけど Lambda@Edge が楽しくなってきた今日このごろ

第 9 章 Takuya Hashimoto / @hasi_t
ぬるめた連載開始うれしい

表紙・ポストカード・ポスター

アベサク

表紙イラスト担当しました。
自由にデザインをさせていただいてありがとうございました。

おが

ブックデザイン担当しました。ご協力いただいた皆様に只々感謝。

木

進行とポストカードイラストで参戦！

ヤマウラ

かっこいい表紙にできたかと思います！

電子版ダウンロード

http://klabgames.tech.blog.jp.klab.com/archives/tbf08.html

KLab Tech Book Vol. 6

2020 年 3 月 7 日　技術書典 8 電子版（1.1）
著　者　KLab 技術書サークル
編　集　水澤 絹子、牧内 大輔
発行所　KLab 技術書サークル
印刷所　日光企画

KLab Tech Book

2020 SPRING Vol.6