# tikz

YI-CHENG LI

November 2023

# 目錄

# 圖目錄

# 程式碼目錄

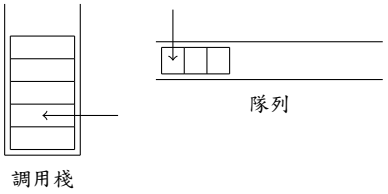
# Chapter 1

# 背景知識

## 1.1 消息隊列

## 1.2 javascript 事件循環

123




## 1.3 Web Workers

javascript 原本的設計是基於單線程實現的，HTML5 提供了 Web Workers API 使 javascript 也可以用運用多線程的能力 javascript 對於異步任務或是平行運算是基於事件循環來實現的，其機制如下

# Chapter 2

# 運算伺服器

運算伺服器是整個繪圖系統的核心，保存著系統當前所有的圖形資料及狀態，根據收到的指令，對圖形進行增刪查改等操作。運算伺服器引用了 CGAL(Computational Geometry Algorithms Library ) 幾何算法庫，用來對圖形進行幾何運算。

幾何運算包含非常多種的演算法，每種演算法所消耗的時間都不一樣，而用戶何時使用何種算法也是不確定的，為了避免整個系統在運算伺服器產生堵塞，需要對其進行解藕設計，在此我們利用 Linux 系統內建的消息隊列來實現，如圖 2.1。

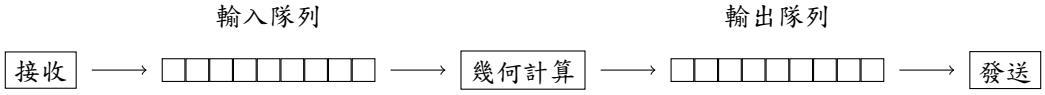



圖 2.1: 運算伺服器的架構

接收、運算、發送，共會用到三個進程，為了便於管理及使用，使用一個主進程啟動，再分裂出三個子進程，如圖 2.2。分裂完成後，主進程進入休眠狀態，當其中一個子進程崩潰時，主進程會被喚醒，進行錯誤處理或是重新產生崩潰的子進程。

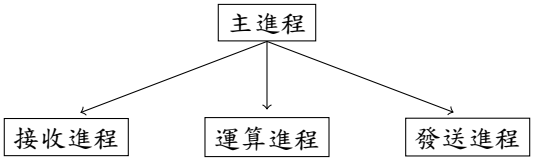



圖 2.2: 運算伺服器的進程分裂機制

```cpp
#include "flow_compute_server.hpp"

int main(){

    pid_t main_process_ID = getpid();
    std::cout << "我是 main_process 進程: " << main_process_ID << std::endl ;


    pid_t rece_process_ID = fork();      //產生接收進程

    if (rece_process_ID == -1) {
        std::cerr << "rece_process fork 失敗" << std::endl;
        return 1;
    }
    else if (rece_process_ID == 0) {
        std::cout << "我是 rece_process 進程: " << getpid() << std::endl ;
        return 0;
    }
    else {
        pid_t comp_process_ID = fork(); //產生運算進程
        if (comp_process_ID == -1) {
            std::cerr << "comp_process fork 失敗" << std::endl;
            return 1;
        }
        else if (comp_process_ID == 0) {
            std::cout << "我是 comp_process 進程: " << getpid() << std::endl ;
            return 0;
        }
        else {
            pid_t send_process_ID = fork();      //產生發送進程
            if (send_process_ID == -1) {
                std::cerr << "send_process fork 失敗" << std::endl;
                return 1;
            }
            else if (send_process_ID == 0) {
                std::cout << "我是 send_process 進程: " << getpid() << std::endl ;
                return 0;
            }
        }
    }
    return 0;
}
```

程式碼 2.1: 運算伺服器的子進程分裂

```
./flow_compute_server
我是 main_process 進程: 18107
我是 rece_process 進程: 18108
我是 comp_process 進程: 18109
我是 send_process 進程: 18110
```

圖 2.3: 程式碼 2.1的輸出結果

# Chapter 3

# 使用者介面

為了能提供跨平台，並且不須額外進行安裝的特性，本系統的使用者介面將在瀏覽器上實現，並且使用最新的 webgpu 技術來繪製圖形。然而，因為 WebGPU 尚未普及，許多平台及瀏覽器都還只支援 WebGL。為了能夠兼容這樣的情況，我們需要同時實作 WebGPU 和 WebGL 兩種繪圖機制，以便在需要時進行切換。

## 3.1 檢查 WebGPU 和 WebGL 是否支援

在網頁加載完成之後，首先需要檢查兩種圖形 API 的支援情況，在此實作兩個檢查函數，並模組化，在需要時引入使用。

```
1
2   //檢查 WebGPU 是否支援的函數
3   export function webgpu_support_check(){
4           if(navigator.gpu){
5                   console.log("WebGPU is support");
6                   return 1;
7           }
8           else{
9                   console.log("WebGPU is not support");
10                  return 0;
11          }
12  }
13
14
15  //檢查 WebGL 是否支援的函數
16  export function webgl_support_check(){
17          if(!!window.WebGLRenderingContext){
18                  console.log("WebGL is support");
19                  return 1;
20          }
21          else{
22                  console.log("WebGL is not support");
23                  return 0;
24
25          }
```

程式碼 3.1: 圖形 API 支援檢查函數