6 Killer function

Javascript

@webdeveloper

1. Debounce Function

A debounce function limits the rate at which a function can fire. This is especially useful for optimizing performance in events like window resizing, scrolling, or keystroke events.

```
function debounce(func, delay) {
  let timeout;
  return function(...args) {
      clearTimeout(timeout);
      timeout = setTimeout(() => func.apply(this, args), delay);
  };
}

// Usage
const log = debounce(() => console.log('Debounced!'), 2000);
window.addEventListener('resize', log);

```

2. Throttle Function

A throttle function ensures that a function is only called at most once in a specified period. This is useful for events that can fire rapidly, such as scrolling or resizing.

```
function throttle(func, limit) {
  let lastFunc;
  let lastRan;
  return function(...args) {
      if (!lastRan) {
          func.apply(this, args);
          lastRan = Date.now();
      } else {
          clearTimeout(lastFunc);
          lastFunc = setTimeout(() => {
              if ((Date.now() - lastRan) >= limit) {
                  func.apply(this, args);
                  lastRan = Date.now();
              }
          }, limit - (Date.now() - lastRan));
      }
  };
}

// Usage
const log = throttle(() => console.log('Throttled!'), 2000);
window.addEventListener('scroll', log);

```

@webdeveloper

3. Currying Function

Currying is a functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

```
function curry(func) {
  return function curried(...args) {
      if (args.length >= func.length) {
          return func.apply(this, args);
      }
      return function(...args2) {
          return curried.apply(this, [...args, ...args2]);
      };
  };
}

// Usage
function sum(a, b, c) {
  return a + b + c;
}
const curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3)); // Output: 6

```

@webdeveloper

4. Memoization Function

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.

```
function memoize(func) {
  const cache = {};
  return function(...args) {
      const key = JSON.stringify(args);
      if (cache[key]) {
          return cache[key];
      }
      const result = func.apply(this, args);
      cache[key] = result;
      return result;
  };
}

// Usage
const fibonacci = memoize(n => (n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)));
console.log(fibonacci(40)); // Output: 102334155

```

5. Deep Clone Function

A deep clone function creates a new object that is a deep copy of the original object, ensuring that nested objects are also copied.

```
function deepClone(obj) {
  return JSON.parse(JSON.stringify(obj));
}

// Usage
const original = { a: 1, b: { c: 2 } };
const cloned = deepClone(original);
cloned.b.c = 3;
console.log(original.b.c); // Output: 2

```

Follow for more

Abhishek Vaghasiya

@webdeveloper