실제로 JavaScript에서 FinalizationRegistry API를 사용하는 경우

끊임없이 진화하는 ECMAScript는 좋지만, 최근에 나오는 일부 기능은 이해하기에 다소 어렵습니다.

특히 가비지 컬렉션(이하 GC)을 위한 새로운 API는 정말 이해하기 어려웠는데, 이를 활용하는 방법을 알아내고 싶었습니다.

이제 실용적인 예제를 통해 FinalizationRegistry API를 사용하는 방법을 알아낼 수 있습니다.

참고사항

WeakSet과 WeakMap은 꽤 오래전부터 사용되어 왔으며, 사용법이 간단합니다. 일반적으로 클래스와 함께 사용하는 것이 합리적입니다. 예를 들어서 클래스의 인스턴스와 관련된 데이터를 추적할 수 있습니다.

```
type PrivateUserData = {
  password: string;
};

const privateUserData = new WeakMap<User, PrivateUserData>();

class User {
  email: string;

  constructor(email: string, password: string) {
    this.email = email;
    privateUserData.set(this, { password });
  }

  changePassword(oldPassword: string, newPassword: string): void {
    const privateData = privateUserData.get(this);

    if (oldPassword !== privateData.password) {
      throw new Error("The current password you entered is incorrect");
    }

    privateData.password = newPassword;
  }
}
```

위의 예시는 실제로 javascript에 private 속성이 없다는 점을 보완하기 위해 WeakMap을 사용하려는 시도입니다. 하지만 이제는 javascript에서 private 속성을 지원합니다.

```
class User {
  email: string;
  #password: string;

  constructor(email: string, password: string) {
    this.email = email;
    this.#password = password;
  }

  changePassword(oldPassword: string, newPassword: string): void {
    if (oldPassword !== this.#password) {
      throw new Error("The current password you entered is incorrect");
    }

    this.#password = newPassword;
  }
}
```

private 속성은 액세스를 방지하면서 데이터를 객체에 연결하는 자연스러운 방법이지만, 위의 두 가지 방법 모두 객체에 대한 참조가 손실되면 데이터가 GC에 의해 수집되어 액세스할 수 없게 되는 동일한 기본 메커니즘에 의존합니다.
당연해 보이지만 이것이 GC을 올바르게 사용하기 위해서는 클래스가 해답이 될 수 있습니다. 예를 들어 다음과 같은 팩토리 함수 스타일을 생각해보세요.

```
type User = {
  getEmail: () => string;
  setEmail: (email: string) => void;
  changePassword: (oldPassword: string, newPassword: string) => void;
};

function createUser(email: string, password: string): User {
  return {
    getEmail: () => email,
    setEmail: (newEmail) => {
      email = newEmail;
    },
    changePassword: (oldPassword, newPassword) => {
      if (oldPassword !== password) {
        throw new Error("The current password you entered is incorrect");
      }

      password = newPassword;
    },
  };
}
```

이 스타일이 인기 있는 이유는 this를 사용하지 않았기 때문입니다. 즉, 렉시컬 스코프 범위 내에 남아있는 로컬 변수(password)가 있습니다. 이것은 이 객체를 분해(destructure)해도 여전히 작동한다는 것을 의미합니다.

```
const { getEmail, setEmail, changePassword } = createUser("", "");
```

this가 없으면 가비지 수집을 위해 직접 추적할 수 있는 개체가 없어집니다.
반면에 prototype과 이 값에 의존하는 javascript 객체는 객체에 대한 참조를 유지해야 합니다.

```
// the "factory" approach -- we don't have to keep the object returned from
// the `createUser` function around
const { getEmail, setEmail } = createUser("", "");
setEmail("foo@bar.baz");
console.log(getEmail()); // "foo@bar.baz"

// the class instance requires us to keep the object in scope
const user = new User("", "");
user.email = "foo@bar.baz";
console.log(user.email);

const { changePassword } = user; // ❌ doesn't work!!
const changePassword = user.changePassword.bind(user);
```

그렇다면 언제 GC에 신경을 써야 할까요?
저는 새로운 FinalizationRegistry API를 실용적인 용도로 사용할 수 있는 방법을 찾기 위해 한참을 고민했고 마침내 현실에서 훌륭한 사용 예시를 발견했습니다.

아마도 여러분은 Promise에 대해 잘 알고 있을 것입니다. 지연(defeereds)에 대해 익숙하시겠지만, 그렇지 않더라도 당황하지 마세요.

일반적으로 javascript에서는 Promise를 반환하는 다른 API를 사용하며, Promise 생성자를 직접 사용하는 경우는 거의 없습니다. 하지만 약간의 창의력을 발휘하면 비동기 이벤트(asynchronous event)를 Promise로 바꿀 수 있습니다. 예를 들어 Promise 생성자를 사용해서 간단한 'sleep'함수를 만들 수 있습니다.

```
function sleep(ms: number): Promise<void> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(), ms);
  });
}
```

이제 복잡한 시나리오를 상상해 보세요. 사용자가 버튼을 클릭하는 등의 다양한 이벤트가 발생할 때 Promise를 반환하는 함수를 만들고 싶지만, Promise가 반환(Resolve)되지 않는다면 어떻게 될까요?

```
const alertModal = document.createElement("div");
alertModal.setAttribute("id", "alertModal");
alertModal.style.display = "none";
alertModal.innerHTML = '<div class="message"></div><button>OK</button>';
document.body.appendChild(alertModal);

const alertMessage = alertModal.querySelector(".message");
const dismissButton = alertModal.querySelector("button");

function alert(message: string): Promise<void> {
  return new Promise<void>((resolve) => {
    alertMessage.innerText = message;
    alertModal.style.display = "block";
    dismissButton.addEventListener("click", onDismiss);

    function onDismiss(): void {
      alertModal.style.display = "none";
      dismissButton.removeEventListener("click", onDismiss);
      resolve();
    }
  });
}
```

여기서 패턴이 보이기 시작합니다. 어떤 비동기 로직이 실행되든 간에, 우리의 코드는 Promise 생성자에 전달된 콜백(Callback) 내에서 호출되어야 하므로 resolve 함수에 접근할 수 있습니다. 이것은 점점 번거로워지기 시작하며, 문제가 될 수 있는 경계에 부딪힐 수도 있습니다. 예를 들어, Reactland에서 비슷한 종류의 이벤트를 기다리는 훅을 만들고 싶다고 가정해 보겠습니다:

```tsx
type AlertContext = {
  visible: boolean;
  open: (message: string) => void;
};

const Context = createContext<AlertContext>(undefined as any);

function AlertProvider({ children }: { children: ReactNode }) {
  const [visible, setVisible] = useState(false);
  const [message, setMessage] = useState<ReactNode>(null);

  const open = useCallback((message: string): void => {
    setVisible(true);
    setMessage(message);
  }, []);

  const close = useCallback((): void => {
    setVisible(false);
  }, []);

  const context = useMemo<AlertContext>(
    () => ({ visible, open }),
    [visible, open],
  );

  return (
    <>
      {children}
      <Modal visible={visible} onClose={close}>
        {message}
      </Modal>
    </>
  );
}

function useAlert(): (message: string) => Promise<void> {
  const { visible, setMessage, setVisible } = useContext(AlertContext);

  const alert = useCallback((message: string): Promise<void> => {
    setVisible(true);
    return new Promise((resolve) => {
      // this is the place where we have access to `resolve`
    });
```

```
  }, [setMessage, setVisible]);

  useEffect(() => {
    if (!visible) {
      // ???
      // this is the place where we want to call `resolve`
    }
  }, [visible]);

  return alert;
}
```

이는 일반적인 패턴이므로 추상화하여 Deferred 패턴을 만드는 것이 합리적입니다.
Deferred 객체는 새로운 Promise를 생성하고 해당 Promise의 Resolve, Reject를 메소드로 제공합니다.

```
class Deferred<T> {
  readonly promise: Promise<T>;
  readonly resolve!: (value: T) => void;
  readonly reject!: (reason?: unknown) => void;

  constructor() {
    this.promise = new Promise<T>((resolve, reject) => {
      this.resolve = resolve;
      this.reject = reject;
    });
  }
}

function alert(message: string): Promise<void> {
  const deferred = new Deferred<void>();

  alertMessage.innerText = message;
  alertModal.style.display = "block";
  dismissButton.addEventListener("click", onDismiss);

  function onDismiss(): void {
    alertModal.style.display = "none";
    dismissButton.removeEventListener("click", onDismiss);
    deferred.resolve();
  }

  return deferred.promise;
}
```

이런 방법으로 Resolve 및 Reject 콜백을 노출하여 React를 사용한다면, "lift up"할 수 있는 단일 인스턴스를 갖게 됩니다.

> lift up은 상위 컴포넌트의 "상태를 변경하는 함수" 그 자체를 하위 컴포넌트로 전달하고, 이 함수를 하위 컴포넌트가 실행하는 것을 의미 합니다.

```
  const { visible, open } = useContext(AlertContext);

  // will wrap the "async operation" that begins when the modal is opened
  // and resolves when user dismisses the modal. it is null when the modal
  // is not open
  const deferred = useRef<Deferred<void> | null>(null);

  useEffect(() => {
    if (!visible) {
      // once dismissed, resolve the promise
      deferred.current?.resolve();
      deferred.current = null;
    }
  }, [visible]);

  const alert = useCallback((message: string): Promise<void> => {
    if (deferred.current) {
      throw new Error("an alert modal is already open");
    }

    // initialize our deferred
    deferred.current = new Deferred();

    // show the modal
    open(message);

    // return the promise
    return deferred.current.promise;
  }, [open]);
```

이 클래스가 현재 이러한 상태로 있는 지금, 사용자가 객체에 대한 참조를 유지할 필요가 없습니다. 우리는 Resolve와 Reject 콜백에 직접 접근할 수 있도록 제공하고 있습니다.

그러나, 우리는 이 클래스를 수정하여 소비자가 참조를 계속 유지하고 구조 분해할 수 없게 만들 수 있습니다. 만약 그렇게 한다면, Deferred 객체가 GC될 때를 알 수 있는 FinalizationRegistry API를 사용할 수 있을지도 모릅니다.

그렇다면 왜 이것이 중요할까요?

Deferred 객체는 감싸진 Promise를 Resolve하거나 Reject하기 위해 접근(Access) 가능해야 합니다.

1. 함수는 deferred 객체를 생성하고,
2. 생성된 Promise를 반환한 다음,
3. 참조를 버려(away the reference) 새로운 Promise를 영구적으로 보류 상태(permanently pending state)로 남겨둘 수 있습니다.
4. 그 Promise를 기다리는 모든 함수는 영원히 기다림에 갇히게 됩니다(stuck waiting forever).

이런 일은 현실에서 발생합니다. 예를 들어, 어떤 이벤트를 기다리는 데 deferred 패턴이 적용된 시나리오를 상상해 보세요. 어쩌면 개발자가 새로운 인증 토큰을 기다린 후 API 요청을 보내는 코드를 작성했을 수 있습니다. 개발자가 예상치 못한 것은 사용자가 새 토큰을 기다리는 동안 "로그아웃"을 클릭하여 요청이 포기되고 페이지의 다른 요소들이 Promise를 계속 기다리게 된다는 것입니다.

Deferred 객체가 GC될 때를 추적할 수 있고, Resolve나 Reject 함수가 GC 전에 호출되지 않았다는 것을 알 수 있다면, 우리는 그 행동을 변경할 수 있습니다. 다른 함수들의 실행을 영원히 중단시키는 대신, 우리는 Promise를 거부할 수 있습니다!

먼저 Deferred 클래스를 수정하여 그것이 충족될 수 없는 상태가 되었을 때를 알 수 있도록 해봅시다.

```
class Deferred<T> {
  readonly #promise: Promise<T>;
  readonly #resolve: (value: T) => void;
  readonly #reject: (reason?: unknown) => void;

  constructor() {
    let resolve!: (value: T) => void;
    let reject!: (reason?: unknown) => void;

    this.#promise = new Promise<T>((res, rej) => {
      res = resolve;
      rej = reject;
    });

    this.#resolve = resolve;
    this.#reject = reject;
  }

  get promise(): Promise<T> {
    return this.#promise;
  }

  resolve(value: T): void {
    this.#resolve(value);
  }

  reject(reason?: unknown): void {
    this.#reject(reason);
  }
}
```

이 변경을 통해 Promise를 Resolve하거나 Reject하는 실제 콜백은 객체의 private 멤버가 됩니다. 실제로 resolve나 reject를 호출하려면, 메소드가 Deferred 클래스 인스턴스에 계속 바인딩되어 있어야 합니다.

이제 까다로운 부분으로 넘어가겠습니다. 우리는 Deferred 객체가 GC될 때 반응할 수 있기를 원합니다. 우리 자신의 코드에서 실수로 영구 참조를 만들어 이 객체가 영원히 메모리에 남아 있도록 하는 일이 없도록 특별히 주의해야 합니다.

먼저, Deferred가 해결되지 않고 GC되었을 때 발생하는 예외를 나타내는 Error 클래스를 만들어 봅시다:

```
class DeferredGarbageCollectedError extends Error {
  readonly name: "DeferredGarbageCollectedError" =
    "DeferredGarbageCollectedError";

  constructor() {
    super(
      "Deferred object was garbage-collected without " +
        "resolving or rejecting the promise",
    );
  }
}
```

FinalizationRegistry API를 사용하면 다른 메타데이터를 객체에 연결할 수 있습니다. 객체에 GC가 발생하면 해당 객체와 연결된 메타데이터와 함께 콜백이 호출됩니다. 따라서 메타데이터는 Promise를 Reject하는 함수여야 합니다. 이 함수는 Deferred 객체에 의존(rely)할 수 없거나 GC 대상이 아니므로 Reject 메서드를 직접 전달할 것입니다.

```
// The metadata ("held value") will be a function to reject the promise
type HeldValue = () => void;
const registry = new FinalizationRegistry<HeldValue>(
  (callback) => callback(),
);

class Deferred<T> {
  readonly #promise: Promise<T>;
  readonly #resolve: (value: T) => void;
  readonly #reject: (reason?: unknown) => void;

  constructor() {
    let resolve!: (value: T) => void;
    let reject!: (reason?: unknown) => void;

    this.#promise = new Promise<T>((res, rej) => {
      res = resolve;
      rej = reject;
    });

    this.#resolve = resolve;
    this.#reject = reject;

    // We register this Deferred instance and provide the callback
    // to reject the promise
    registry.register(
      this, // the value we are tracking for GC
      () => reject(new DeferredGarbageCollectedError()),
        // the "held value" references locally scoped `reject`
        // rather than `this.#reject` so it has no dependency on
        // `this` and won't prevent GC
    );
  }

  get promise(): Promise<T> {
    return this.#promise;
  }

  resolve(value: T): void {
    this.#resolve(value);

    // When resolve or reject are called, we no longer care
    // about garbage collection
```

```
      registry.unregister(this);
    }

    reject(reason?: unknown): void {
      this.#reject(reason);

      // When resolve or reject are called, we no longer care
      // about garbage collection
      registry.unregister(this);
    }
  }
```

이것이 실제로 효과가 있는지 알아보기 위해 간단한 데모 애플리케이션을 만들었습니다. 직접 확인해 보세요!

데모