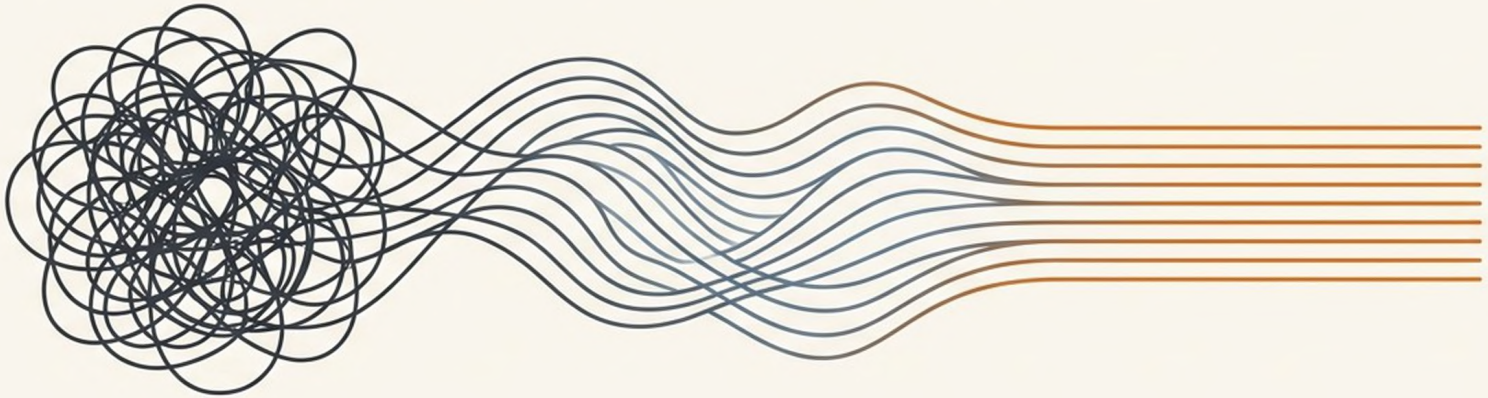
Укрощение асинхронности в JavaScript

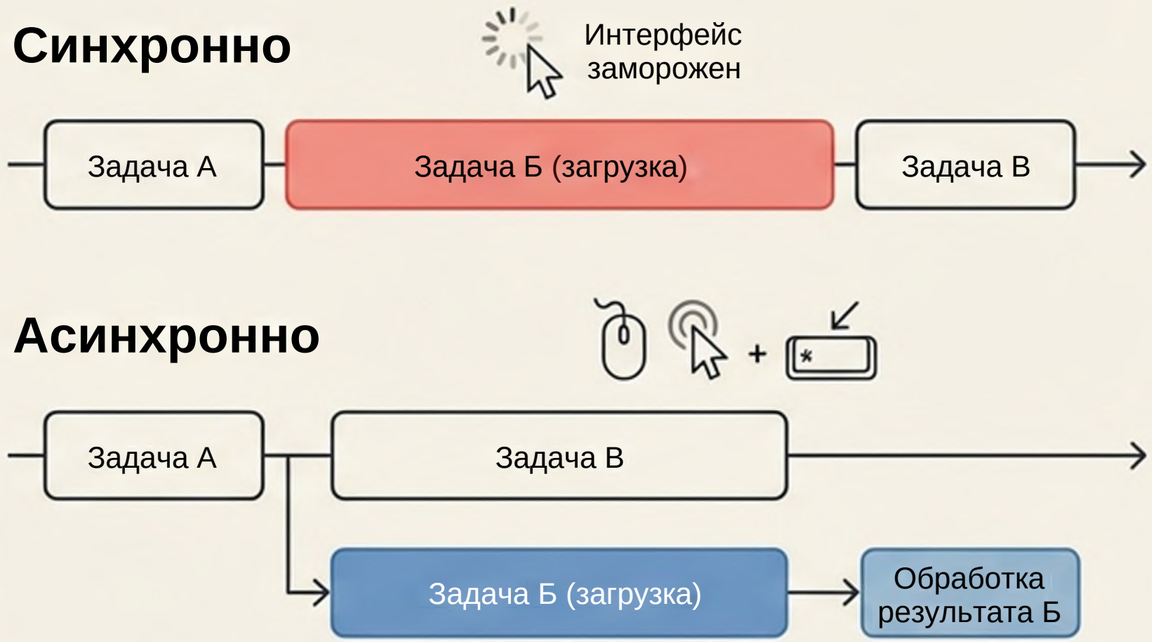
Путь от хаоса колбэков к элегантности 'async/await'



Эта презентация — это история решения одной из фундаментальных проблем JavaScript. Мы проследим эволюцию асинхронного кода, чтобы понять не только как работают современные подходы, но и почему они появились.

Проблема: асинхронные операции в однопоточном мире

JavaScript выполняет код в одном потоке. Если задача (например, сетевой запрос) занимает много времени, она блокирует всё остальное, делая интерфейс неотзывчивым. Асинхронность позволяет выполнять долгие задачи в фоновом режиме, не останавливая основной поток.




```
// Эта функция загружает скрипт асинхронно.
// Действие (загрузка) будет завершено не сейчас, а потом.
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('/my/script.js');
// Код ниже не будет ждать загрузки скрипта.
```

Первое решение: функция обратного вызова (колбэк)

Чтобы узнать, когда асинхронная операция завершилась, мы передаём ей функцию — «колбэк». Эта функция будет вызвана с результатом (или ошибкой) по окончании действия.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  // Вызываем callback(null, script) при успехе
  script.onload = () => callback(null, script);

  // Вызываем callback(error) при ошибке
  script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

  document.head.append(script);
}

// Использование:
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен, можно использовать его функции
    newFunction();
  }
});
```

> **Ключевая идея:** Это подход «колбэк с первым аргументом-ошибкой» («error-first callback») — стандартная практика в асинхронном программировании старой школы.

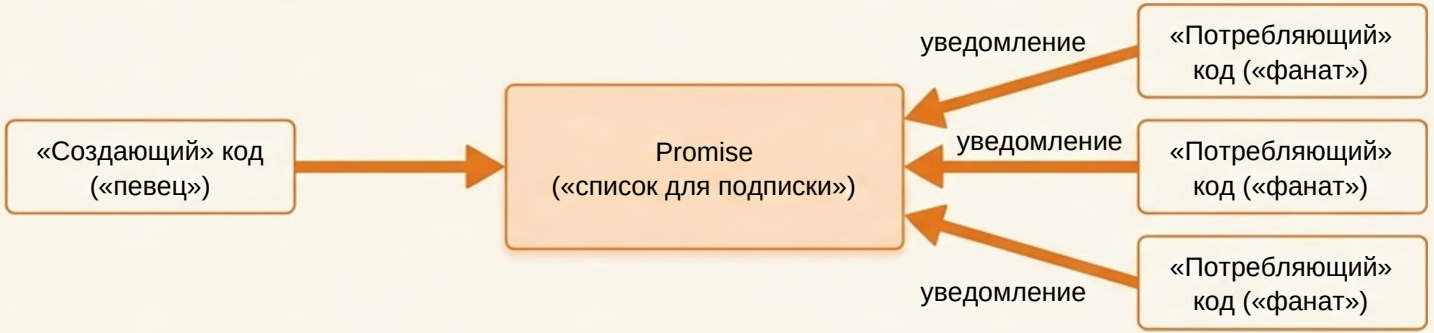
Проблема разрастается: «Адская пирамида вызовов»

Когда требуется выполнить несколько асинхронных действий последовательно, каждое новое действие вкладывается в колбэк предыдущего. Код растёт вправо, превращаясь в трудночитаемую и хрупкую структуру.

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...и так далее, пока все скрипты не будут загружены
          }
        });
        }
      })
    }
});
```

Новая надежда: объект Promise

Промис — это специальный объект, который «обещает» предоставить результат в будущем. Он связывает **«создающий» код** (который выполняет асинхронную операцию) и **«потребляющий» код** (который ждёт результата).



Аналогия «Певец и фанаты»

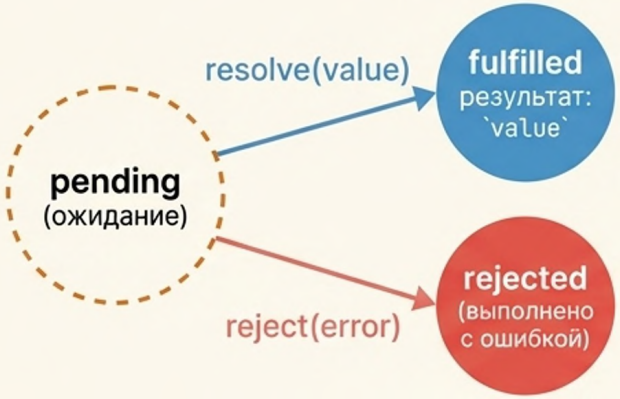
1. **«Создающий» код («певец»):** Занимается долгой работой (запись сингла).
2. **«Потребляющий» код («фанаты»):** Хочет получить результат, как только он будет готов.
3. **Promise («список для подписки»):** Певец даёт фанатам список, куда они могут записаться. Когда сингл выйдет, все подписчики получат уведомление.

Синтаксис:

```
let promise = new Promise(function(resolve, reject) {
    // функция-исполнитель (executor)
    // Внутри неё выполняется асинхронный код.

    // По завершении нужно вызвать один из колбэков:
    // resolve(value) - если всё успешно
    // reject(error) - если произошла ошибка
});
```

Жизненный цикл промиса: три состояния

У каждого промиса есть внутреннее состояние. Изначально это 'pending', а затем — одно из двух конечных состояний. Переход возможен только один раз.



state: "fulfilled"
(выполнено успешно)
result: value

state: "rejected"
(выполнено с ошибкой)
result: error

Важный факт: Учитывается только первый вызов resolve или reject. Все последующие вызовы игнорируются. Это гарантирует, что результат будет только один.

```
let promise = new Promise(function(resolve, reject) {
    resolve("успех!");

    reject(new Error("...")); // будет проигнорирован
    setTimeout(() => resolve("...")); // тоже проигнорирован
});
```

Потребление промисов: '.then', '.catch' и '.finally'

Методы '.then', '.catch' и '.finally' позволяют «подписаться» на результат промиса. Они регистрируют функции, которые будут вызваны, когда промис изменит своё состояние.

Сравнение "До" и "После"

Слева (Колбэки)

```
// Было:
loadScript('/my/script.js', function(err, script) {
    if (err) {
        alert('Ошибка: ${err.message}');
    } else {
        alert('Скрипт загружен!');
    }
});
```

Справа (Промисы)

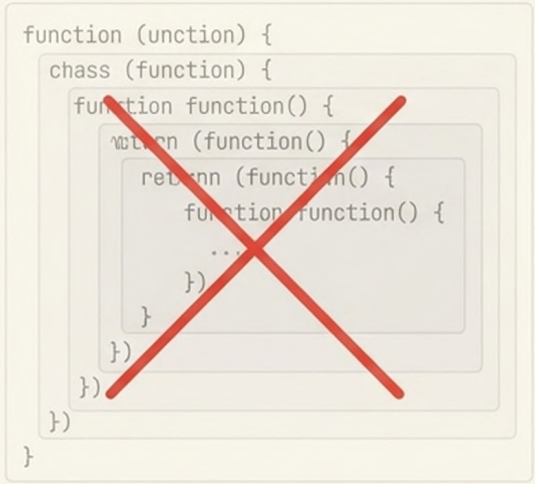
```
// Стало:
function loadScript(src) {
    return new Promise((resolve, reject) => {
        /* ...логика loadScript... */
        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error(/*...*/));
    });
}

loadScript('/my/script.js')
    .then(script => alert('Скрипт загружен!'))
    .catch(error => alert('Ошибка: ${error.message}'));
```

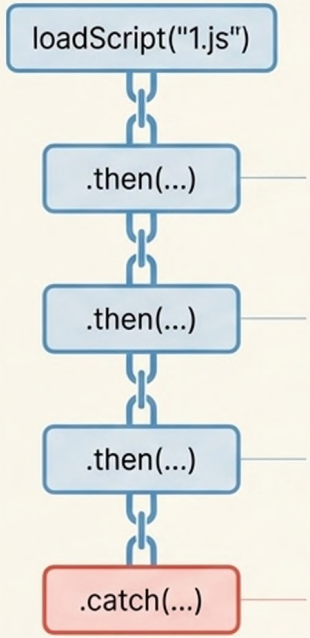
`'.finally(f)': Вызывается в любом случае — при успехе или ошибке. Идеально подходит для очистки, например, для скрытия индикатора загрузки.

Главная сила: цепочки промисов (Chaining)

Каждый вызов .then() возвращает новый промис. Это позволяет выстраивать последовательные асинхронные операции в элегантную, плоскую цепочку, которая легко читается сверху вниз.

Слева (Колбэки)



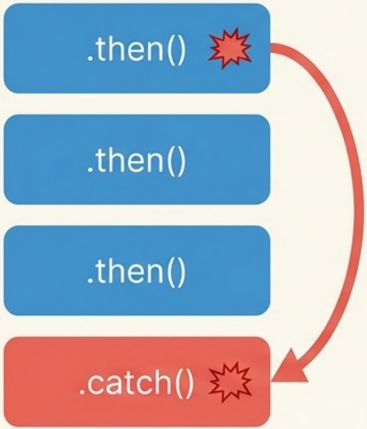


****Вертикальный, читаемый поток****

```
loadScript("1.js")
 .then(script => {
    console.log('Первый скрипт загружен');
    return loadScript("2.js"); // Возвращаем новый промис
})
 .then(script => {
    console.log('Второй скрипт загружен');
    return loadScript("3.js");
})
 .then(script => {
    console.log('Третий скрипт загружен');
    // все скрипты загружены последовательно
})
 .catch(error => {
    console.error('Произошла ошибка при загрузке', error);
});
```

Надёжная обработка ошибок

Ошибка в любом месте цепочки (явный 'reject' или исключение 'throw') «проваливается» вниз до ближайшего обработчика '.catch()'. Это как невидимый 'try...catch', оборачивающий каждый шаг.

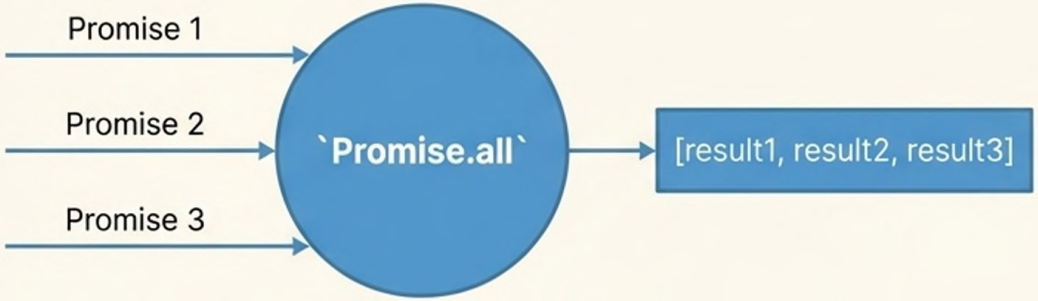



```
fetch('/article/promise-chaining/user.json') // 1. Запрос данных
    .then(response => response.json()) // 2. Парсинг JSON
    .then(user => {
        // ... что-то делаем с пользователем
        blabla(); // 3. Упс! Несуществующая функция -> исключение!
    })
    .then(githubUser =>{/*... этот шаг будет пропущен ... */ })
    .catch(error => {
        // 4. Ошибка из шага 3 будет поймана здесь
        alert(error); // ReferenceError: blabla is not defined
    });
```

Инструментарий: параллельное выполнение с 'Promise.all'

Используется, когда нужно дождаться выполнения *_всех_* промисов. Если хотя бы один из них завершится с ошибкой, весь '**Promise.all'** немедленно завершается с этой ошибкой. Идеально для сценариев «всё или ничего».

Сценарий: Загрузить шаблон, стили и данные перед рендерингом страницы.



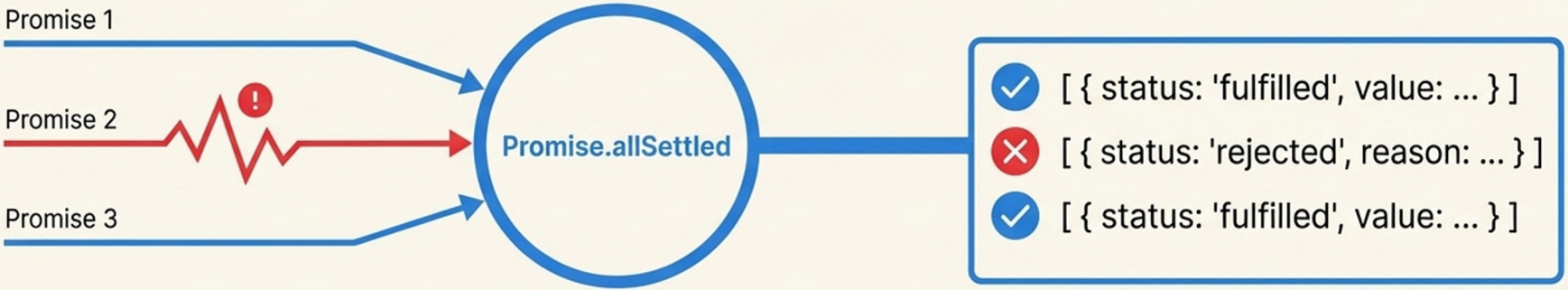

```
let urls = [
    'https://api.github.com/users/iliakan',
    'https://api.github.com/users/remy',
    'https://api.github.com/users/jeresig'
];

let requests = urls.map(url => fetch(url));

Promise.all(requests)
    .then(responses => {
     // responses — массив объектов ответа
     for(let response of responses) {
         alert('${response.url}: ${response.status}');
     }
});
```

Инструментарий: получение всех результатов с 'Promise.allSettled'

В отличие от **'Promise.all'**, этот метод всегда ждёт завершения *всех* промисов, независимо от их исхода. Возвращает массив объектов, описывающих результат каждого промиса.

Сценарий: Загрузить информацию о нескольких пользователях. Если один запрос не удался, мы всё равно хотим получить данные по остальным.

```
let urls = [
    'https://api.github.com/users/iliakan',
    'https://no-such-url',
    'https://api.github.com/users/jeresig'
];

Promise.allSettled(urls.map(url => fetch(url)))
    .then(results => {
    results.forEach((result, num) => {
        if (result.status == "fulfilled") {
            alert('${urls[num]}: ${result.value.status}');
        }
        if (result.status == "rejected") {
            alert('${urls[num]}: ${result.reason}');
        }
    });
});
```



Высшая форма: элегантность 'async/await'

Это специальный синтаксис для работы с промисами, который позволяет писать асинхронный код так, будто он синхронный. Это «синтаксический сахар» над промисами, а не их замена.

async: Ставится перед функцией. Означает, что функция *всегда* возвращает промис.

await: Работает только внутри 'async' функций. Заставляет JavaScript ждать, пока промис справа от него не выполнится, после чего возвращает его результат.

Простой пример:

```
async function f() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("готово!"), 1000)
    });
    let result = await promise; // выполнение "паузы" до разрешения промиса
    alert(result); // "готово!"
}

f();
```

Важно: 'await' не блокирует весь движок. Пока он ждёт, JavaScript может выполнять другие скрипты и обрабатывать события.

Трансформация кода: '.then/catch' против 'async/await'

Давайте перепишем пример с загрузкой аватара пользователя с GitHub, чтобы увидеть разницу. Код становится линейным, чистым и интуитивно понятным.

Слева (Цепочка промисов)

```
function showAvatar() {
  return fetch('/user.json')
    .then(response => response.json())
    .then(user => fetch(
        'https://api.github.com/users/user/${user.name}'))
    .then(response => response.json())
    .then(githubUser => new Promise(resolve => {
      let img = document.createElement('img');
      img.src = githubUser.avatar_url;
      document.body.append(img);
      setTimeout(() => {
        img.remove();
        resolve(githubUser);
      }, 3000);
    }));
}
```

Справа ('async/await')

```
async function showAvatar() {
  let response = await fetch('/user.json');
  let user = await response.json();

  let githubResponse = await fetch('https://api.github.com/users/user/${user.
  let githubUser = await githubResponse.json();

  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  document.body.append(img);

  await new Promise(resolve => setTimeout(resolve, 3000));

  img.remove();
  return githubUser;
}
```

Знакомая обработка ошибок с 'try...catch'

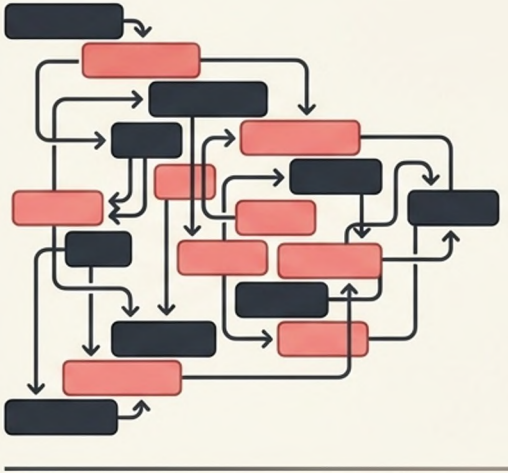
Если промис, которого мы 'await', завершается с ошибкой, 'await' выбрасывает эту ошибку. Это позволяет нам использовать для обработки ошибок стандартную и хорошо знакомую конструкцию 'try...catch'.

```
async function loadJson(url) {
  try {
    let response = await fetch(url);

    if (response.status == 200) {
      return await response.json();
    } else {
      // Выброшенная ошибка будет поймана в блоке catch
      throw new Error(response.status);
    }
  } catch (err) {
    // Перехватывает ошибки как из fetch, так и из throw
    alert(err);
  }
}

loadJson('no-such-user.json');
```

Сравнение: Это гораздо нагляднее, чем .catch(), особенно при сложной логике обработки ошибок.

Эволюция асинхронного JavaScript

КОЛБЭКИ



Сложность и низкая читаемость

ПРОМИСЫ

Структура и предсказуемость

ASYNC/AWAIT

```
async function getData() {
  const result = await fetch(url);
  ...
}
```

Ясность и элегантность

Ключевые выводы и современный подход

Колбэки порождают «пирамиду ада» и сложны в поддержке. Их следует избегать в новом коде.

Промисы — это фундаментальный строительный блок для асинхронных операций. Они привнесли структуру, цепочки вызовов и надёжную обработку ошибок.

→ **'async/await' —** это современный стандарт. Он предоставляет чистый, синхронно-подобный синтаксис для написания и чтения кода, основанного на промисах.

> **Золотое правило:**
> Всегда используйте 'async/await' в новом коде для максимальной читаемости. Но всегда помните и понимайте, что под капотом у него работают промисы.