

КАК СТАТЬ АВТОРОМ



Неделя тестировщиков

Налетай, торопись: мы тут собрали темы для статей, которые ждёт сообщество



WOLFRIEND 18 июня в 17:50

## Делаем отзывчивый и максимально возможный размер шрифта динамического текста относительно контейнера

CSS\*, JavaScript\*, HTML\*, ReactJS\*, Дизайн

Из песочницы

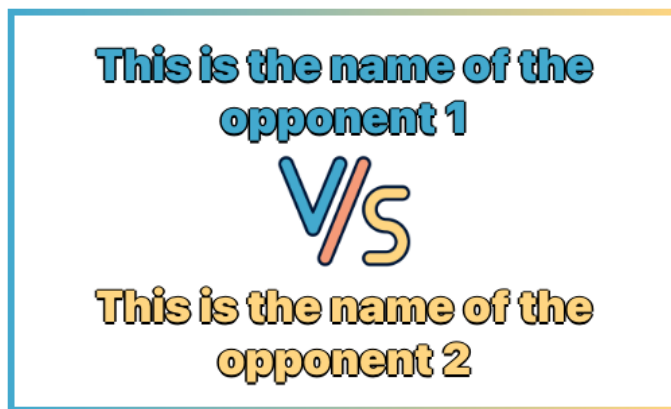
Перед нами часто возникает задача, сделать текст отзывчивым в зависимости от размера экрана устройства. Казалось бы, задача вполне тривиальна, и сходу можно назвать несколько вариантов её решения, не ломая голову, но всегда есть дополнительные условия, которые усложняют выполнение простых задач. В данной статье мы будем рассматривать решение небольшой задачи: как сделать максимально возможный размер шрифта динамического текста в его родительском контейнере. Или же, как впихнуть невпихуемое.

### Предисловие

Для простоты, все условия задачи, реализация, способы применения решение и остальные не относящиеся к делу детали (такие, как стили) будут максимально упрощены, для того, что б сконцентрироваться на требуемом результате. Как и где правильно применить данное решение, зависит только от вас и вашей ситуации. Приведенные ниже примеры кода будут с использованием React и TypeScript, но их знание совсем не обязательно, знаний нативного JavaScript будет вполне чем достаточно.

### Задача

Представьте карточку или баннер, в котором мы должны отображать текст, получаемый с бекенда. Мы не знаем заранее какой длинны будет текст, но мы хотим сделать размер шрифта максимально возможным. Что бы текст был не мелким и читаемым (в случае если это будет всего лишь одно слово), и для того, что б он не вылезал за рамки своего контейнера (в случае если слов будет множество). Помимо этого, мы заранее не знаем размеров контейнера, в котором будет располагаться наш текст. Он может быть фиксированным, например с заранее заданными размерами ширины и высоты каким-либо администратором в CMS системе, или же наоборот, это может быть полностью отзывчивый контейнер, который расстанется на всю ширину и высоту родителя. Для более реального кейса, мы будем делать баннер с двумя именами состоящих между собой соперников. Примерно это будет выглядеть следующим образом:



Готовый баннер.



Демонстрация работы (нажмите на картинку для анимации).

Разметка баннера максимально простая, это контейнер содержащий в себе 3 элемента - текст №1, картинка и текст №2 которые будут располагаться друг над другом. Внутри каждого текстового контейнера будет располагаться тег `<p>`.

```
<div className="banner">
  <div className="banner__text-container">
    <p className="banner__text banner__text_1">{TEXT_1}</p>
  </div>
  <img className="banner__img" src={icon} />
  <div className="banner__text-container">
    <p className="banner__text banner__text_2">{TEXT_2}</p>
  </div>
</div>
```

## Возможные варианты, которые были отброшены

Ниже будет список тех вариантов, которые рассматривались как возможное решение, но не подошли по тем или иным причинам.

1. [Media queries](#) - Первое, что пришло в голову - это использование медиа выражений. Эта идея отпала быстро по причине динамической длины текста, и огромного количества всевозможных комбинаций.
2. `clamp()` - Неплохой и довольно-таки гибкий способ, который в большинстве случаев может сгодиться. На эту тему даже есть неплохая статья: [Linearly Scale font-size with CSS clamp\(\) Based on the Viewport](#) с рабочей "песочницей", где можно просчитать необходимые для себя размеры. Но, "поигравшись" с данной песочницей, и перепробовав различные варианты (строго установленные размеры баннера, или наоборот никаких размеров), возникали случаи, где текст ломался и выходил за рамки.
3. SVG `<text>` - Широко рекомендуемый вариант, но заметно прожорливый при ресайзинге и довольно сложный в настройке, в случае когда мы мало что знаем о размерах (длина текста, размеры контейнера).
4. Сторонние библиотеки и плагины - этот вариант был отброшен сразу, что б не засорять проект лишними зависимостями.
5. Подсчёт количества букв - Была и такая безумная идея, вычислять размеры контейнера в котором находится текст, подсчитывать количество букв получаемого текста, и на основании этого делать расчёты размера шрифта и возможного количества строк. Сложность в

вычислениях, входящий текст может быть на разных языках (проблема подсчёта букв), и с различными символами (проблема поиска целого слова).

---

## Решение

Логика решения довольно проста. И вкратце состоит из нескольких шагов:

1. Находим необходимые текстовые узлы.
2. Вычисляем размеры родительского контейнера (высоту и ширину).
3. Пытаемся подобрать максимально возможный размер шрифта, что б текст не вылезал за рамки родительского контейнера.

Последний пункт звучит немного "костыльным", как будто мы пытаемся методом тыка попасть в нужную нам точку. Отчасти так и есть, но мы не пытаемся глупо добавлять по `1px`, а делаем это более элегантным путём, что б сократить количество ненужных вычислений и уменьшить нагрузку.

---

## Структура

Структура максимально простая и понятная с использованием [npm create-react-app](#). В некоторых местах используются стили, для более приятного визуального восприятия.

```
├─ package-lock.json
├─ package.json
├─ public
│   └─ index.html
├─ src
│   ├── App.css
│   ├── App.tsx
│   ├── components
│   │   └─ Banner
│   │       ├── Banner.scss
│   │       └─ Banner.tsx
│   ├── constants
│   │   └─ index.ts
│   ├── hooks
│   │   └─ useStretchingText.ts
│   ├── index.css
│   ├── index.js
│   └─ static
│       └─ vs.png
└─ tsconfig.json
```

---

## Описание решения

Пройдёмся в целом по структуре проекта и реализации решения.

В корневом компоненте `App.tsx` расположен главный `Banner.tsx`, который циклом проходит по переменной `OPPONENTS` отрисовывая значения различной длины для наглядности.

```
import React from "react";
```

```
import './App.scss';

import { Banner } from './components/Banner/Banner';

import { OPPONENTS } from './constants';

export const App = () => {
  return (
    <div className="app">
      {OPPONENTS.map((banner, index) => (
        <Banner key={index} {...banner} />
      ))}
    </div>
  );
};
```

Основной `Banner.tsx` также максимально прост, в нём содержится простая разметка, которая содержит имя первого оппонента, иконку, и имя второго оппонента. И как уже можно заметить, функцию которая и отвечает за отзывчивость текста, но об этом немного попозже.

```
import './Banner.scss';

import { useStretchingText } from '../../../hooks/useStretchingText';

const icon = require('../../../static/vs.png');

export const Banner = ({ text_1, text_2 }) => {
  useStretchingText("banner__text");

  return (
    <div className="banner">
      <div className="banner__text-container">
        <p className="banner__text banner__text_1">{text_1}</p>
      </div>
      <img className="banner__img" src={icon} />
      <div className="banner__text-container">
        <p className="banner__text banner__text_2">{text_2}</p>
      </div>
    </div>
  );
};
```

Немного о стилях `Banner.scss`, в большинстве своём они тут всего лишь для красоты, но некоторые моменты я бы хотел объяснить.

1. Размеры иконки заданы явно для простоты.
2. Высота баннера также явно задана, но это не обязательно. В оригинальной задаче, мой баннер должен был размещаться поверх другого контейнера как дополнительный слой, с абсолютным позиционированием.
3. Размеры каждого текстового контейнера (где располагается тег `<p>` с именем оппонента) вычисляются равномерно, вычитая размеры картинки, паддинги.

#### ► Стили

Далее начинается самое интересное, реализация решения вынесена в кастомный хук (для тех кто не работал с React - воспринимайте это как обычную функцию) `useStretchingText.ts`. Буду проходить по нему построчно и объяснять что там происходит. Хук принимает два аргумента:

1. `textClassName` - Это класс, по которому мы будем находить необходимый текст.
2. `initialMinFontSize` - Изначально минимально допустимый размер шрифта. На случай если мы не хотим допустить значение ниже установленного.

Инициализируем переменные которые будем использоваться позже:

```
let fontSize: number,  
    maxHeight: number,  
    maxWidth: number,  
    parentElement: HTMLElement,  
    maxCycles: number = 100;
```

Находим текстовые ноды по указанному классу:

```
const textElements: NodeList = document.querySelectorAll(  
    .${textClassName}  
);
```

Делаем проверку на наличие найденных текстовых узлов, и проходимся по каждому из них:

```
if (textElements.length) {  
    textElements.forEach((element) => {  
        // Some logic...  
    });  
}
```

Делаем некоторые вычисления, на основании родительского контейнера (с классом `.banner__text-container`), рассчитываем максимально возможные размеры контейнера с текстом (с классом `.banner__text`), устанавливаем минимальный и максимальный размеры шрифта. Для ясности хочу сказать, что в условиях задачи нам нужно найти **максимальный** размер шрифта, но в логике решения мы будем отталкиваться от **минимального** значения. То есть, не ниже которого, наши требования не будут удовлетворены. После получения значений, применяем стиль к текстовому узлу.

```
parentElement = element.parentElement;  
maxWidth = parentElement.clientWidth;  
maxHeight = parentElement.clientHeight;  
fontSize = maxHeight;  
let minFontSize = initialMinFontSize;  
let maxFontSize = fontSize;  
element["style"].fontSize = `${fontSize}px`;
```

После того как мы применили первый размер шрифта, происходит вычисление и проверка, удовлетворяет ли текущее значение наши требования, и так до тех пор, пока не будет найдено оптимальное значение. Пару слов о происходящем: Цикл продолжается до тех пор, пока текущий размер шрифта не равняется минимально удовлетворяющему размеру шрифта. Если высота и ширина текстового узла (с классом `.banner__text`) меньше или равны максимально допустимым, значит мы меняем значение `minFontSize`, если же наоборот, мы вылезли за допустимые рамки, мы меняем значение `maxFontSize`. После чего мы рассчитываем новое вероятно подходящее для нас значение размера шрифта. Но, что б не делать лишних вычислений, прибавляя по `1px`, мы делаем это по следующей формуле:

***fontSize = Math.floor((minFontSize + maxFontSize)/2);***

В дополнении к этому, мы обезопасили себя заранее с помощью переменной, которую мы объявляли выше `maxCycles: number = 100;`. Чтобы по каким-либо причинам не

"провалиться" в бесконечный цикл, мы прервём вычисления, достигнув установленного лимита.

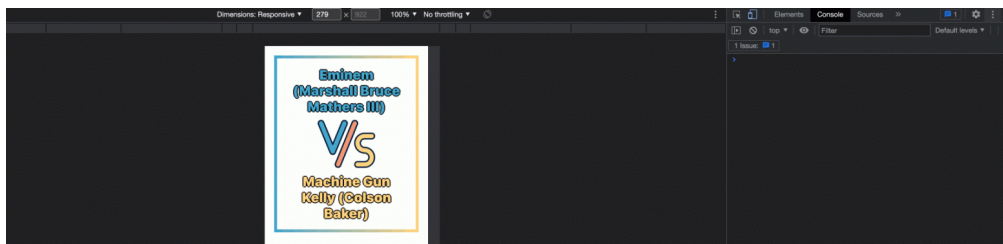
```
while (fontSize !== minFontSize) {
  element["style"].fontSize = `${fontSize}px`;

  if (
    element["offsetHeight"] <= maxHeight &&
    element["offsetWidth"] <= maxWidth
  ) {
    minFontSize = fontSize;
  } else {
    maxFontSize = fontSize;
  }

  fontSize = Math.floor((minFontSize + maxFontSize) / 2);

  --maxCycles;
  if (maxCycles <= 0) {
    console.error("The maximum cycle exceeded");
    break;
  }
}
```

Ниже на анимации показано, что происходит если задать слишком маленькое значение для `maxCycles`: `number = 100`; . Это не сломает работу, но также и не даст нам зависнуть на бесконечных вычислениях.



Маленькое значение `maxCycles` (нажмите на картинку для анимации).

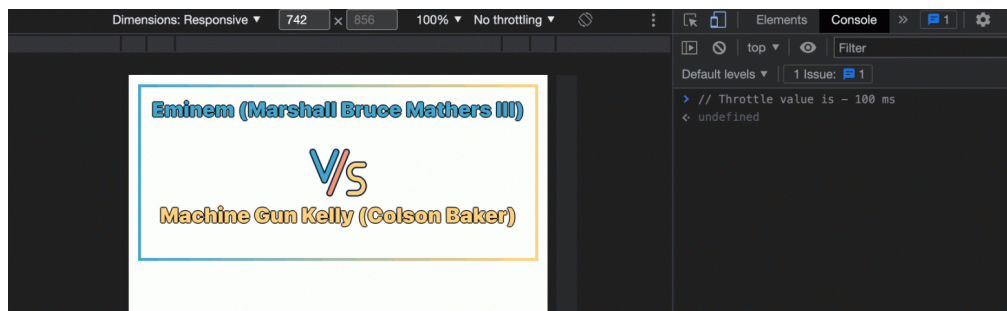
После того как подходящий размер шрифта найден, применяем конечный стиль к элементу, и делаем то же самое со следующим.

```
element["style"].fontSize = `${minFontSize}px`;
```

Ещё один небольшой приём оптимизации, с помощью использования функции `throttle`, библиотеки `Lodash`. С её помощью, мы будем ограничивать максимальное количество вызовов функции в заданный интервал. Я использую значение `15`, так как где-то встречал информацию, что браузеры производят перерисовку контента страницы на ранее чем каждые 16 миллисекунд.

```
const debouncedFunction = throttle(stretchingText, 15);
```

Если установить слишком большое значение для `throttle` функции (в нашем случае 100 мс уже является большим значением), то будут явно заметны задержки в работе:



Большое значение throttle (нажмите на картинку для анимации).

Далее, я использую слушатель на изменение размера экрана, для выполнения функции, но данный подход индивидуален в каждом случае, и должен быть использован на ваше усмотрение. Для целей демонстрации его вполне достаточно:

**P.S. Навешивание прослушивателя таким образом является не совсем хорошим решением и в данном случае и сделано исключительно для показания работоспособности.**

```
useEffect(() => {
  window.addEventListener("resize", debouncedFunction);
  debouncedFunction();

  return () => {
    window.removeEventListener("resize", debouncedFunction);
  };
}, []);
```

Конечный результат готового хука выглядит так:

```
import { useEffect } from "react";
import { throttle } from "lodash";

export const useStretchingText = (
  textClassName: string,
  initialMinFontSize = 3
): void => {
  const stretchingText = () => {
    let fontSize: number,
        maxHeight: number,
        maxWidth: number,
        parentElement: HTMLElement,
        maxCycles: number = 50;

    const textElements: NodeList = document.querySelectorAll(
      `.${textClassName}`
    );

    if (textElements.length) {
      textElements.forEach((element) => {
        parentElement = element.parentElement;
        maxWidth = parentElement.clientWidth;
        maxHeight = parentElement.clientHeight;
        fontSize = maxHeight;

        let minFontSize = initialMinFontSize;
        let maxFontSize = fontSize;

        element["style"].fontSize = `${fontSize}px`;

        while (fontSize !== minFontSize) {
          element["style"].fontSize = `${fontSize}px`;
        }
      });
    }
  };
  stretchingText();
};
```



```

    if (
      element["offsetHeight"] <= maxHeight &&
      element["offsetWidth"] <= maxWidth
    ) {
      minFontSize = fontSize;
    } else {
      maxFontSize = fontSize;
    }

    fontSize = Math.floor((minFontSize + maxFontSize) / 2);

    --maxCycles;
    if (maxCycles <= 0) {
      console.error("The maximum cycle exceeded");
      break;
    }
  }

  element["style"].fontSize = `${minFontSize}px`;
});
}
};

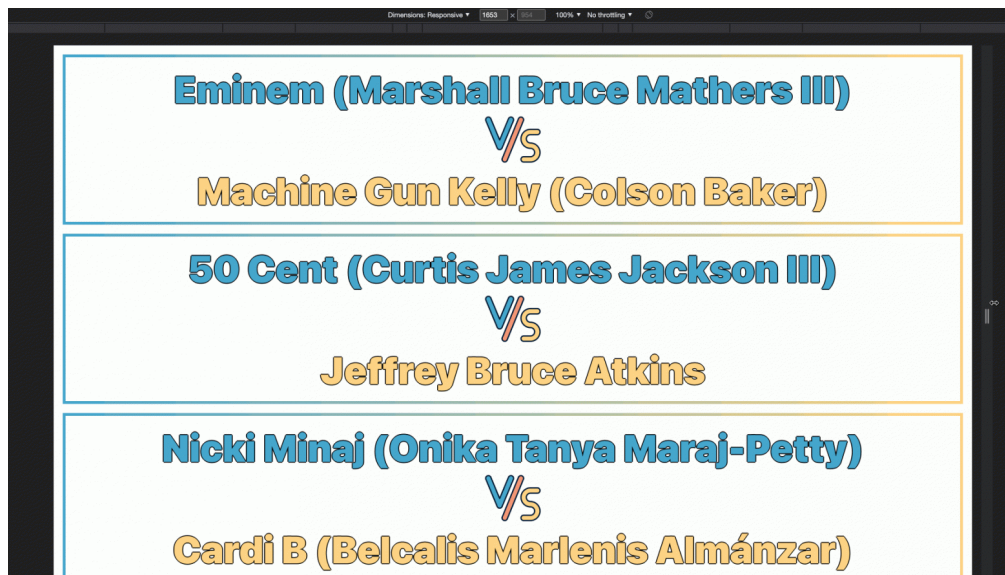
const debouncedFunction = throttle(stretchingText, 15);

useEffect(() => {
  window.addEventListener("resize", debouncedFunction);
  debouncedFunction();

  return () => {
    window.removeEventListener("resize", debouncedFunction);
  };
}, []);
};

```

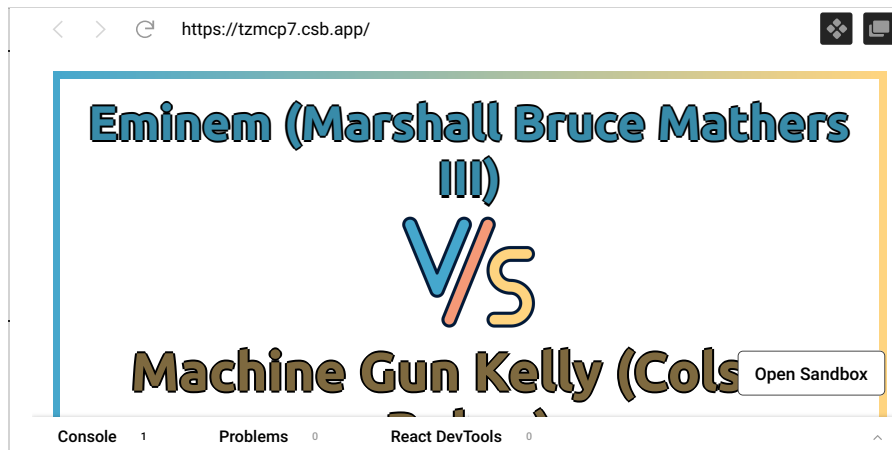
Демонстрация работы:



Демонстрация работы (нажмите на картинку для анимации).

Песочница:





## Что можно было бы улучшить

1. Кеширование. Возможно реализовать какое-то простое кеширование, которое будет запоминать последнее (или несколько последних) вычисленных значений, и применять к остальным элементам. Это хорошо подходит для нашего баннера, так как контейнеры обоих оппонентов абсолютно равны, и вычислив значение для одного, нам не нужно производить те же операции для другого.
2. Использование хука и подписок. Сейчас это сделано в качестве кастомного хука и добавлением `addEventListener` на ресайзинг страницы, но возможно это и не пригодится, достаточно вызывать функцию лишь при изменении ориентации экрана.
3. Глобальное использование. Возможен вариант когда данное вычисление придётся применить глобально ко всему проекту, соответственно вызов функции нужно будет вынести куда-то на верхний уровень.
4. Ограничения размеров. У нас уже есть минимально допустимый размер шрифта, как опциональный второй аргумент `initialMinFontSize`. Вполне вероятно, что кому-то понадобится также добавить максимально допустимое значение, на случай если текст может состоять всего из одного слова, а сам баннер будет растянут на весь экран.
5. Согласованность. В данный момент, каждый текстовый узел рассчитывает свои стили отдельно. Это значит, что каждый текст будет иметь свой размер шрифта. И если на примере нашего баннера имя первого оппонента будет состоять всего из одного слова, а имя второго из множества слов, разница в размере шрифта может быть значительной. Это не минус, а всего лишь один из способов решения. Возможно кому-то нужно иметь единый размер шрифта для всех оппонентов баннера (основываясь на меньшем), а кому-то наоборот, каждый оппонент должен иметь свой размер шрифта. Это всего лишь предложение, которое стоит взять во внимание.

## Полезные материалы

1. Код проекта: [GitHub](#).
2. Онлайн песочница: [Codesandbox](#).


**Теги:** [javascript](#), [markup](#), [font](#), [responsive](#), [react.js](#), [css](#), [html](#), [markdown](#)

**Хабы:** [CSS](#), [JavaScript](#), [HTML](#), [ReactJS](#), [Дизайн](#)



Комментарии 8 


7/4/22, 13:37

◦  **Sap.ru**  
18.06.2022 в 23:37

Но необходимый размер текста можно вычислить после первой же итерации!

◆ 0 Ответить



◦  **dom1n1k**  
19.06.2022 в 03:50

Кажется, смысл вашей статьи можно уместить в 3 строчки:

1. Аналитического решения задачи не существует, будем использовать перебор.
2. Для оптимизации полный перебор заменим на бинарный поиск.
3. При ресайзе вьюпорта используем тротлинг.

Зачем тут все эти стены кода — не совсем понятно.

И, кстати, тротлинг и дебонсинг это не одно и то же.

◆ +1 Ответить



◦  **sunnybear**  
19.06.2022 в 16:19

Аналитическое решение (оно же кэширование с условиями) существует. Но, конечно, всегда проще переложить вычисления из собственной головы на браузер пользователя.

◆ 0 Ответить



◦  **demimurych**  
20.06.2022 в 20:36

Простите если я чего ляпну не так.

Первое на что хотел бы обратить внимание это variable fonts. Ведь там, набор параметров, позволяющих регулировать отображение шрифта просто гиганский.

И второе. Ваша задача уже лет 10 как решена в Google Docs. Причем решена двумя способами.

Способ первый заключается в том, что фактически все шрифты с которыми работает Docs парсятся, или заранее имеют предустановленные метрики глифов. Что дает возможность очень точно рассчитывать все длины, зная размеры каждой буквы.

Способ второй - тот самый svg. Который они используют уже почти год вполне уверенно. И с точки зрения решения только этой проблемы, делают они это очень неплохо. Возможно вам стоит заглянуть в их canvas render, что бы забрать от туда этот кусок кода.

◆ 0 Ответить



◦  **PaulZi**  
23.06.2022 в 05:54

1) Размер шрифта можно легко вычислить через `measureText`

2) Ну а самый простой вариант, это делать все же в svg text, и одной строчкой кода просто корректировать viewBox, согласно длине `text.getBBox()`.

◆ 0 Ответить



Только полноправные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

#### ПОХОЖИЕ ПУБЛИКАЦИИ

11 мая в 12:38

**Фронтенд-новости №5. Отказ от React, новые CSS-свойства для адаптивности и JS-контейнеры**

◆ +7

👁 14K

📖 41

💬 5 +5

17 марта в 12:22

**1С http сервис с jwt авторизацией + клиент React js**

◆ +3

👁 3.2K

📖 17

💬 7 +7

22 декабря 2021 в 17:19

## React.js: размышления об управлении состоянием и повторном рендеринге

♦ +5

👁 9.7K

📖 57

💬 18 +18

МИНУТОЧКУ ВНИМАНИЯ

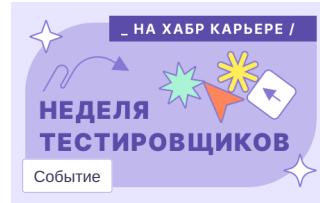
Разместить



Жизненный квест про работу в банке



Промокод — твой билет в общество потребления



Неделя тестировщиков на Хабр Карьере

### ВОПРОСЫ И ОТВЕТЫ

Можно ли как то получить элемент из двух одинаковых, но у одного есть доп класс?

HTML · Простой · 0 ответов

Функция должна принимать другую функцию и возвращать результат вызова этой функции. Вызов функции неверный?

JavaScript · Простой · 1 ответ

Как прикреплять меню, когда до него доскролят?

CSS · Средний · 1 ответ

Как удалить div 2-го родителя div'a в котором Checkbox?

JavaScript · Средний · 1 ответ

JS: Можно ли посмотреть ссылки на объект?

JavaScript · Простой · 0 ответов

Больше вопросов на Хабр Q&A

### ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 11:16

#### Bash отладчик с поддержкой произвольных точек останова

♦ +44

👁 4.1K

📖 74

💬 5 +5

вчера в 14:40

#### Новые нули дзета-функции

♦ +31

👁 4K

📖 14

💬 10 +10

вчера в 21:20

#### Личный опыт выгорания

♦ +26

👁 8K

📖 31

💬 23 +23

вчера в 17:00

#### DIY квантовые вычисления: как я начал собирать квантовые схемы

♦ +21

👁 8.2K

📖 41

💬 22 +22

вчера в 21:00

#### Антиматерия и бариогенезис. Три причины, почему нет антивещества, но есть мы

+15

2.8K

21

1 +1

Про котиков, биохакинг или U-Boot? Какие статьи ждут читатели Хабра

Турбо

Реклама



ЧИТАЮТ СЕЙЧАС

Эксперт пояснил, что обнаруженная «Персеверанс» сброшенная кожа ксеноморфа — это часть «Небесного крана»

39K 39 +39

Завершился международный изоляционный эксперимент SIRIUS-21 длиной 240 суток

12K 17 +17

Украинский изобретатель собрал рабочий «математический велосипед»

25K 44 +44

Личный опыт выгорания

8K 23 +23

Теория выученной беспомощности. Что это и откуда она взялась

3.9K 23 +23

Как устроен МКБ Реактор: история одного стартапа

Мегапост

РАБОТА

Ваш аккаунт

Разделы

Информация

Услуги

Войти

Публикации

Устройство сайта

Корпоративный блог

Регистрация

Новости

Для авторов

Медийная реклама

Хабы

Компании

Авторы

Песочница

Для компаний

Документы

Соглашение

Конфиденциальность

Нативные проекты

Образовательные программы

Мегaproекты



Настройка языка

Техническая поддержка

Вернуться на старую версию

© 2006–2022, Habr

проект и попытка стресс-тест для посещений за рубеж или по России

🚩 Реклама Hybrid 0