Your web app is taking up too much RAM

Let's fix it!

@giuliozausa
How much of that is actually JS data?

<table>
<thead>
<tr>
<th>Size</th>
<th>Speed</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>672 kB</td>
<td>7.7 kB/s</td>
<td>sw-desktop_v4.js</td>
</tr>
<tr>
<td>106 MB</td>
<td>1.1 kB/s</td>
<td><a href="http://www.airbnb.com">www.airbnb.com</a>: Main</td>
</tr>
<tr>
<td>111 MB</td>
<td>2.3 kB/s</td>
<td>Total JS heap size</td>
</tr>
<tr>
<td>#</td>
<td>Game Number</td>
<td>&quot;Game Length&quot;</td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>February 9, 2024</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>February 21, 2024</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>February 22, 2024</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>February 24, 2024</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>February 26, 2024</td>
</tr>
<tr>
<td>6</td>
<td>29</td>
<td>February 28, 2024</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>February 30, 2024</td>
</tr>
<tr>
<td>8</td>
<td>117</td>
<td>March 2, 2024</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
<td>March 4, 2024</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>March 6, 2024</td>
</tr>
<tr>
<td>11</td>
<td>40</td>
<td>March 8, 2024</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>March 10, 2024</td>
</tr>
</tbody>
</table>

*7.9 MB*

*1421 MB*
16 channel relay module

Description:
16 channel relay that will allow you to control different types of electrical loads

Created:
June 17th 2022

Last updated by: jhanwinbarrozo
3 months ago
1 Contributor(s)

Properties

Availability & Pricing

Add a Manufacturer Part Number to see inline results.

- Mouser
- Digi-Key
- LCCS
Is faster enough?
Why We Memo All the Things

October 28, 2020

On my team at Coinbase, we ask everyone to use the React performance trinity — `memo`, `useMemo`, and `useCallback` — all the time. For some reason, this is controversial. I’m guessing this has something to do with Twitter. This article explains why we do it anyway.

Why We React.memo All Components

Let’s start with what we can all agree on: in most apps, some components can benefit from being wrapped in `React.memo`. Maybe because they are expensive to rerender, or maybe they are children of a component that renders much more frequently. Maybe both.

So not using `memo` at all is not an option. We are left with two options:

- Use `memo` some of the time
- Use `memo` all the time

The first option sounds like the most appealing, doesn’t it? Figure out when we can benefit from `React.memo`, and use it then, and only then. However, before we go that far, we have to remind ourselves that we work on a large team. No matter how diligent we are with education, code review, and profiling, we are not going to get it right 100% of the time. So we have to ask ourselves:
Why Optimise Memory Usage?
Out Of Memory Crashes
Notion mobile app and browser crash reboot loop - unusable

Notion crashes (goes into a crash/reboot loop) on mobile app and browser with a database of more than about 50-75 rows/pages. We've tested this on multiple phones. All do it - although some work ok when others are crashing. Then tomorrow the users that worked yesterday, crash today. Works good on a computer. This is unacceptable. I've submitted support tickets and so far all i get is a response that they had issues yesterday or the other day but resolved them. I have a team of 6 people having this issue. So bad it is unusable on mobile.
Long Garbage Collection times
Users multitask
How do we solve this?

1. Identify what occupies memory 🕵
2. Kill it with fire 🔥
3. Make sure we don't repeat the same mistake 🚨
How do we solve this?

1. **Identify what occupies memory** 🕵
2. Kill it with fire 🔥
3. Make sure we don't repeat the same mistake 🚨
Static or Transient?
Static or **Transient?**
Count vs Size

500MB

130MB
Count vs Size

...can become hundreds of MBs!
Shallow vs Retained Size

Shallow: 40 bytes
Retained: 40 bytes + 1MB x 10 = 10 MB
Allocation Types

- Code: 29,505 kB
- Strings: 21,791 kB
- JS arrays: 115,077 kB
- Typed arrays: 42,278 kB
- System objects: 2,337 kB
- Total: 699,817 kB
Tooling 🛠
Chrome Memory Profiler
Heap Snapshots
Let's play with the memory profiler
Select profiling type

- Heap snapshot
  Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.
  - Include numerical values in capture

- Allocation instrumentation on timeline
  Allocation timelines show instrumented JavaScript memory allocations over time. Once profile is recorded you can select a time interval to see objects that were allocated within it and still alive by the end of recording. Use this profile type to isolate memory leaks.
  - Record stack traces of allocations (extra performance overhead)

- Allocation sampling
  Record memory allocations using sampling method. This profile type has minimal performance overhead and can be used for long running operations. It provides good approximation of allocations broken down by JavaScript execution stack.

Select JavaScript VM instance

<table>
<thead>
<tr>
<th>Size</th>
<th>Usage Rate</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>164 MB</td>
<td>611 kB/s</td>
<td><a href="http://www.flux.ai">www.flux.ai</a>: Main</td>
</tr>
<tr>
<td>944 kB</td>
<td>2.4 kB/s</td>
<td>service-worker.js</td>
</tr>
<tr>
<td>74.7 MB</td>
<td>52.0 kB/s</td>
<td>pbcl.Layout Engine Worker.bee7f5c7db7738a4581f.worker.js</td>
</tr>
<tr>
<td>40.7 MB</td>
<td>5.4 kB/s</td>
<td>UserCodeRuntime.worker.eaa63fa9447107c8689.worker.js</td>
</tr>
<tr>
<td>2.7 MB</td>
<td></td>
<td>td.doubleclick.net: 347414527</td>
</tr>
</tbody>
</table>

283 MB | 671 kB/s | Total JS heap size

Start | Load
// We keep all the useFrames in a map with uuids, so we can keep track of all of them removing the old ones
const fastUseFrameEvents = new Map<string, FrameCallback>();

export function useFrameFast(fn: FrameCallback) {
  // UUID created only the first time
  const [myId] = useState(() => uuid());

  // On unmount we clear the record
  useEffect(() => void fastUseFrameEvents.delete(myId), [myId]);

  // At every render we have a new callback, so we update it (potentially dangerous with concurrent mode?)
  fastUseFrameEvents.set(myId, fn);
}
On a 375 parts document there is almost a 50% improvement.

Before (the browser almost crashed because of an OOM):
Those are simply too many!
<table>
<thead>
<tr>
<th>#</th>
<th>Game Number</th>
<th>&quot;Game Length&quot;</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td></td>
<td>February 9, 2024</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td></td>
<td>February 21, 2024</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td></td>
<td>February 24, 2024</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Heap Snapsorts

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Distance</th>
<th>Shallow Size</th>
<th>Retained Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object x7784789</td>
<td>2</td>
<td>230 640 456</td>
<td>407 230 396</td>
</tr>
<tr>
<td>Object @35665843</td>
<td>11</td>
<td>488 0 %</td>
<td>15 940 0 %</td>
</tr>
<tr>
<td>Object @34181177</td>
<td>11</td>
<td>472 0 %</td>
<td>6 960 0 %</td>
</tr>
<tr>
<td>Object @35842959</td>
<td>11</td>
<td>464 0 %</td>
<td>11 156 0 %</td>
</tr>
<tr>
<td>Object @4687219</td>
<td>11</td>
<td>456 0 %</td>
<td>4 856 0 %</td>
</tr>
<tr>
<td>Object @35665273</td>
<td>11</td>
<td>456 0 %</td>
<td>6 324 0 %</td>
</tr>
<tr>
<td>Object @34591097</td>
<td>11</td>
<td>436 0 %</td>
<td>16 912 0 %</td>
</tr>
<tr>
<td>Object @38590855</td>
<td>11</td>
<td>432 0 %</td>
<td>19 728 0 %</td>
</tr>
<tr>
<td>Object @35211917</td>
<td>11</td>
<td>432 0 %</td>
<td>19 728 0 %</td>
</tr>
<tr>
<td>Object @34182579</td>
<td>11</td>
<td>456 0 %</td>
<td>6 324 0 %</td>
</tr>
<tr>
<td>Object @29747185</td>
<td>13</td>
<td>428 0 %</td>
<td>15 412 0 %</td>
</tr>
<tr>
<td>Object @36053603</td>
<td>8</td>
<td>412 0 %</td>
<td>20 808 0 %</td>
</tr>
<tr>
<td>Object @33117210</td>
<td>10</td>
<td>398 0 %</td>
<td>5 992 0 %</td>
</tr>
</tbody>
</table>
memlab

Analyzes JavaScript heap and finds memory leaks in browser and node.js

Define Your Test

Define E2E test scenarios on browser interaction:

```javascript
// test.js
function url() {
    return 'https://www.google.com/maps/place/Sili
}
async function action(page) {
    await page.click('button[aria-label="Hotels"]'
}
async function back(page) {
    await page.click('[aria-label="Close"]');
}
module.exports = {action, back, url};
```

Run memlab in CLI

Find memory leaks with the custom E2E test scenario:

```
$ memlab run --scenario test.js
```

Support memory analyses for the previous browser test:

```
# Analyze duplicated string in heap
$ memlab analyze string
# Check unbound object growth
$ memlab analyze unbound-object
# Get shapes with unbound growth
$ memlab analyze unbound-shape
# Discover more memory analyses
$ memlab analyze -h
```

Programming API

Memory analysis for JavaScript heap snapshots:

```
const {findLeaks, takeSnapshots} = require('@memlab/snapshots');
async function test() {
    const scenario = {
        url: () => 'https://www.facebook.com',
    };
    const result = await takeSnapshots(scenario);
    const leaks = findLeaks(result);
    // ...
}
```
Powerful API for snapshots!

class MyAnalysisTest extends BaseAnalysis {
  override getCommandName(): string {
    return "my_analysis";
  }

  override getFilename(): string {
    return "Example Analysis";
  }

  override async process(options: HeapAnalysisOptions): Promise<void> {
    const snapshotPath = pluginUtils.getAbsolutePathForAnalysis(options);
    const snapshot = await utils.getSnapshotFromFile(snapshotPath, {buildNodeIdIndex: true, verbose: true});
    analysis.preparePathFinder(snapshot);

    // Do stuff here with the snapshot!
  }

  export default MyAnalysisTest;
Which *types of objects* are taking up the most space, out of the 2 millions we found in the snapshot?

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Distance</th>
<th>Shallow Size</th>
<th>Retained Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object @x2254628</td>
<td>2</td>
<td>62 534 152</td>
<td>199 263 952</td>
</tr>
<tr>
<td>Object @6415517</td>
<td>9</td>
<td>12 0 %</td>
<td>16 939 416</td>
</tr>
<tr>
<td>Object @5447681</td>
<td>4</td>
<td>12 0 %</td>
<td>10 888 816</td>
</tr>
<tr>
<td>Object @3086099</td>
<td>6</td>
<td>28 0 %</td>
<td>5 190 988</td>
</tr>
<tr>
<td>Object @3803509</td>
<td>6</td>
<td>12 0 %</td>
<td>3 277 032</td>
</tr>
<tr>
<td>Object @2900085</td>
<td>6</td>
<td>12 0 %</td>
<td>2 174 664</td>
</tr>
<tr>
<td>Object @4066881</td>
<td>9</td>
<td>164 0 %</td>
<td>1 282 884</td>
</tr>
<tr>
<td>Object @4866883</td>
<td>9</td>
<td>12 0 %</td>
<td>1 280 500</td>
</tr>
<tr>
<td>Object @4842229</td>
<td>8</td>
<td>36 0 %</td>
<td>1 225 608</td>
</tr>
<tr>
<td>Object @12782069</td>
<td>9</td>
<td>28 0 %</td>
<td>1 224 440</td>
</tr>
<tr>
<td>Object @4067157</td>
<td>7</td>
<td>196 0 %</td>
<td>1 221 428</td>
</tr>
<tr>
<td>Object @5338541</td>
<td>7</td>
<td>16 0 %</td>
<td>1 190 544</td>
</tr>
<tr>
<td>Object @5345999</td>
<td>10</td>
<td>12 0 %</td>
<td>1 181 120</td>
</tr>
<tr>
<td>Object @1179777</td>
<td>19</td>
<td>28 0 %</td>
<td>1 095 972</td>
</tr>
<tr>
<td>Object @10423241</td>
<td>10</td>
<td>12 0 %</td>
<td>1 088 576</td>
</tr>
<tr>
<td>Object @3158795</td>
<td>6</td>
<td>12 0 %</td>
<td>935 548</td>
</tr>
<tr>
<td>Object @5342799</td>
<td>7</td>
<td>40 0 %</td>
<td>888 980</td>
</tr>
</tbody>
</table>
1. Load the Snapshot

2. Find all the object types

3. Compute total shallow size for each type

4. Sort and print results
Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
Array: 24.9MB (11.8%)
FiberNode: 23.3MB (11.05%)
string: 15.3MB (7.26%)
Object { current }: 13.5MB (6.43%)
Object { create, deps, destroy, next }: 13MB (6.16%)
Vector3: 5MB (2.39%)
Object { }: 4.5MB (2.15%)
Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
Object { baseQueue, next, queue }: 4.2MB (2.02%)
Vector2: 3.7MB (1.79%)
Group: 2.6MB (1.26%)
Matrix4: 2.5MB (1.2%)
Object { context, memoizedValue, next }: 2.5MB (1.19%)
Euler: 1.9MB (0.93%)
Quaternion: 1.8MB (0.89%)
Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
LineCurve: 1.7MB (0.81%)
HyperInstantingPlaceholder: 1.6MB (0.8%)
BufferGeometry: 1.4MB (0.67%)
Object { dispatch, interleaved, lastRenderedReducer, lastRenderedState, pending }: 1.3MB (0.65%)
Object { lastEffect, stores }: 1.3MB (0.65%)
Object { $typeof, key, props, ref, type, ... }: 1.2MB (0.6%)
Object { firstContext }: 1.1MB (0.55%)
Mesh: 1MB (0.51%)
CirclePlaceholder: 1MB (0.48%)
Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
Array: 24.0MB (11.6%)
FiberNode: 23.3MB (11.05%)
string: 15.3MB (7.26%)
Object { current }: 13.5MB (6.43%
Object { create, deps, destroy, next }: 13MB (6.16%)
Vector3: 5MB (2.39%)
Object { }: 4.5MB (2.15%)
Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
Object { baseQueue, next, queue }: 4.2MB (2.02%)
Vector2: 3.7MB (1.79%)
Group: 2.6MB (1.26%)
Matrix4: 2.5MB (1.2%)
Object { context, memoizedValue, next }: 2.5MB (1.19%)
Euler: 1.9MB (0.93%)
Quaternion: 1.8MB (0.89%)
Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
LineCurve: 1.7MB (0.81%)
HyperInstanceingPlaceholder: 1.6MB (0.8%)
BufferGeometry: 1.4MB (0.67%)
Object { dispatch, interleaved, lastRenderedReducer, lastRenderedState, pending }: 1.3MB (0.65%)
Object { lastEffect, stores }: 1.3MB (0.65%)
Object { $$typeof, key, props, ref, type, ... }: 1.2MB (0.6%)
Object { firstContext }: 1.1MB (0.55%)
Mesh: 1MB (0.51%)
CirclePlaceholder: 1MB (0.48%)
export type Hook = {
  memoizedState: any,
  baseState: any,
  baseQueue: Update<any, any> | null,
  queue: any,
  next: Hook | null,
};
Keeping track of hooks is expensive!

Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
FiberNode: 23.3MB (11.05%)
string: 15.3MB (7.26%)
Object { current }: 13.5MB (6.43%)
Object { create, deps, destroy, next }: 13MB (6.16%)
Vector3: 5MB (2.39%)
Object { }: 4.5MB (2.15%)
Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
Object { baseQueue, next, queue }: 4.2MB (2.02%)
Vector2: 3.7MB (1.79%)
Group: 2.6MB (1.26%)
Matrix4: 2.5MB (1.2%)
Object { context, memoizedValue, next }: 2.5MB (1.19%)
Euler: 1.9MB (0.93%)
Quaternion: 1.8MB (0.89%)
Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
LineCurve: 1.7MB (0.81%)
HyperInstancingPlaceholder: 1.6MB (0.8%)
1. Find all the FiberNode data structures in memory
2. Determine which React component they belongs to
3. Compute statistics about that FiberNode
4. Accumulate all the computed statistics, grouping them by React component type
SuperDuperPathInstance:
  instances: 671
  total: 1.1MB
  shallow: 88.5KB
  total children: 0 byte
  total sibling: 18.7KB
  total memoizedProps: 190.4KB
  total memoizedState: 864.5KB
  per hook:
  [0]  40.2KB (useMemo)
  [1]  40.2KB (useMemo)
  [2]  53.6KB (useMemo)
  [3]  53.6KB (useMemo)
  [4]  48.3KB (useMemo)
  [5]  69.7KB (useMemo)
  [6]  10.7KB (useRef)
  [7]  10.7KB (useRef)
  [8]  42.9KB (useMemo)
  [9]  67.1KB (useMemo)
  [10] 75.1KB (useMemo)
  [11] 40.2KB (useMemo)
  [12] 96.6KB (useLayoutEffect)

Portal:
  instances: 696
  total: 4.1MB
  shallow: 91.8KB
  total children: 0 byte
  total sibling: 19.4KB
  total memoizedProps: 183.8KB
  total memoizedState: 353.5KB
  per hook:
  [0]  261.6KB (useContext)
  [1]  30.6KB (useState)
  [2]  61.2KB (useState)
  [3]  879.6KB (useCallback)
  [4]  62.6KB (useState)
  [5]  61.2KB (useEffect)

AbstractRouteSegment:
  instances: 1340
  total: 277.6KB
  shallow: 176.8KB
  total children: 0 byte
  total sibling: 37.5KB
  total memoizedProps: 112.5KB
  total memoizedState: 4.5MB
  per hook:
  [0]  85.7KB (useMemo)
  [1]  85.7KB (useContext)

@715773:
  instances: 19828
  total: 27.1MB
  shallow: 2.6MB
  total children: 0 byte
  total sibling: 999.5KB
  total memoizedProps: 5.5MB
  total memoizedState: 19.3MB
  per hook:
  [0]  160 bytes (useContext)
  [1]  160 bytes (useRef)
  [2]  704 bytes (useRef)
  [3]  768 bytes (useImperativeHandle)
How many **strings** are UUIDs?

```javascript
let totalSize = 0;
let uuidSize = 0;
snapshot.nodes.forEach((node) => {
  if (node.type === "string") {
    const matchesUuid =
      /^[0-9a-fA-F]{8}\b-\[0-9a-fA-F\]{4}\b-\[0-9a-fA-F\]{4}\b-\[0-9a-fA-F\]{4}\b-\[0-9a-fA-F\]{12}.+$/.exec(node.name,
        );
    if (matchesUuid) {
      // console.log(node.name);
      uuidSize += node.self_size;
    }
    totalSize += node.self_size;
  }
});

info.topLevel(`Total size of strings: ${utils.getReadableBytes(totalSize)}`);
info.topLevel(`Total size of UUIDs: ${utils.getReadableBytes(uuidSize)}`);
```
Memory Analysis is Difficult
But, for some apps, it makes the difference

Notion crashes (goes into a crash/reboot loop) on mobile app and browser with a database of more than about 50-75 rows/pages. We've tested this on multiple phones. All do it - although some work ok when others are crashing. Then tomorrow the users that worked yesterday, crash today. Works good on a computer. This is unacceptable. I've submitted support tickets and so far all I get is a response that they had issues yesterday or the other day but resolved them. I have a team of 6 people having this issue. So bad it is unusable on mobile.
The Chrome Profiler is cool...
...but sometimes is not enough
Thank you!

@giuliozausa