Breaking Changes in Version 10

- `cache.fetch()` return type is now `Promise<V | undefined>` instead of `Promise<V | void>`. This is an irrelevant change practically speaking, but can require changes for TypeScript users.

  For more info, see the change log.

lru-cache

A cache object that deletes the least-recently-used items.

Specify a max number of the most recently used items that you want to keep, and this cache will keep that many of the most recently accessed items.

This is not primarily a TTL cache, and does not make strong TTL guarantees. There is no preemptive pruning of expired items by default, but you *may* set a TTL on the cache or on a single `set`. If you do so, it will treat expired items as missing, and delete them when fetched. If you are more interested in TTL caching than LRU caching, check out @isaacs/ttlcache.

As of version 7, this is one of the most performant LRU implementations available in JavaScript, and supports a wide diversity of use cases. However, note that using some of the features will necessarily impact performance, by causing the cache to have to do more work. See the “Performance” section below.

Installation

```
npm install lru-cache --save
```

Usage

```
// hybrid module, either works
import { LRUCache } from 'lru-cache'
// or:
const { LRUCache } = require('lru-
        cache')
// or in minified form for web browsers:
import { LRUCache } from 'http://
        unpkg.com/lru-cache@9/dist/mjs/
        index.min.mjs'

// At least one of 'max', 'ttl', or
        'maxSize' is required, to
        prevent
// unsafe unbounded storage.
//
// In most cases, it's best to specify a
        max for performance, so all
// the required memory allocation is
        done up-front.
//
// All the other options are optional,
        see the sections below for
// documentation on what each one does.
        Most of them can be
// overridden for specific items in
        get()/set()
const options = {
  max: 500,
```


If you were relying on the internals of LRUCache in version 6 or before, it probably will not work in version 7 and above.

Breaking Changes in Version 8

- The `fetchContext` option was renamed to `context`, and may no longer be set on the cache instance itself.
- Rewritten in TypeScript, so pretty much all the types moved around a lot.
- The AbortController/AbortSignal polyfill was removed. For this reason, **Node version 16.14.0 or higher is now required**.
- Internal properties were moved to actual private class properties.
- Keys and values must not be `null` or `undefined`.
- Minified export available at `'lru-cache/min'`, for both CJS and MJS builds.

Breaking Changes in Version 9

- Named export only, no default export.
- AbortController polyfill returned, albeit with a warning when used.

If performance matters to you:

1. If it's at all possible to use small integer values as keys, and you can guarantee that no other types of values will be used as keys, then do that, and use a cache such as lru-fast, or mnemonist's LRUCache which uses an Object as its data store.
2. Failing that, if at all possible, use short non-numeric strings (ie, less than 256 characters) as your keys, and use mnemonist's LRUCache.
3. If the types of your keys will be anything else, especially long strings, strings that look like floats, objects, or some mix of types, or if you aren't sure, then this library will work well for you.

If you do not need the features that this library provides (like asynchronous fetching, a variety of TTL staleness options, and so on), then mnemonist's LRUMap is a very good option, and just slightly faster than this module (since it does considerably less).

4. Do not use a `dispose` function, size tracking, or especially ttl behavior, unless absolutely needed. These features are convenient, and necessary in some use cases, and every attempt has been made to make the performance impact minimal, but it isn't nothing.

Breaking Changes in Version 7

This library changed to a different algorithm and internal data structure in version 7, yielding significantly better performance, albeit with some subtle changes as a result.


```
// for use with tracking overall
      storage size
maxSize: 5000,
sizeCalculation: (value, key) => {
  return 1
},

// for use when you need to clean up
      something when objects
// are evicted from the cache
dispose: (value, key, reason) => {
  freeFromMemoryOrWhatever(value)
},

// for use when you need to know that
      an item is being inserted
// note that this does NOT allow you
      to prevent the insertion,
// it just allows you to know about
      it.
onInsert: (value, key, reason) => {
  logInsertionOrWhatever(key, value)
},

// how long to live in ms
ttl: 1000 * 60 * 5,

// return stale items before removing
      from cache?
allowStale: false,

updateAgeOnGet: false,
updateAgeOnHas: false,
```

```
  // async method to use for
        cache.fetch(), for
  // stale-while-revalidate type of
        behavior
  fetchMethod: async (key, staleValue,
        { options, signal, context })
        => {},
}

const cache = new LRUCache(options)

cache.set('key', 'value')
cache.get('key') // "value"

// non-string keys ARE fully supported
// but note that it must be THE SAME
        object, not
// just a JSON-equivalent object.
var someObject = { a: 1 }
cache.set(someObject, 'a value')
// Object keys are not toString()-ed
cache.set('[object Object]', 'a
        different value')
assert.equal(cache.get(someObject), 'a
        value')
// A similar object with same keys/
        values won't work,
// because it's a different object
        identity
assert.equal(cache.get({ a: 1 }),
        undefined)

cache.clear() // empty the cache
```


Note that coercing *anything* to strings to use as object keys is unsafe, unless you can be 100% certain that no other type of value will be used. For example:

```
const myCache = {}
const set = (k, v) => (myCache[k] = v)
const get = k => myCache[k]

set({}, 'please hang onto this for me')
set('[object Object]', 'oopsie')
```

Also beware of "Just So" stories regarding performance. Garbage collection of large (especially: deep) object graphs can be incredibly costly, with several "tipping points" where it increases exponentially. As a result, putting that off until later can make it much worse, and less predictable. If a library performs well, but only in a scenario where the object graph is kept shallow, then that won't help you if you are using large objects as keys.

In general, when attempting to use a library to improve performance (such as a cache like this one), it's best to choose an option that will perform well in the sorts of scenarios where you'll actually use it.

This library is optimized for repeated gets and minimizing eviction time, since that is the expected need of a LRU. Set operations are somewhat slower on average than a few other options, in part because of that optimization. It is assumed that you'll be caching some costly operation, ideally as rarely as possible, so optimizing set over get would be unwise.

If you need to track `undefined` values, and still note that the key is in the cache, an easy workaround is to use a sigil object of your own.

```
import { LRUCache } from 'lru-cache'
const undefinedValue =
        Symbol('undefined')
const cache = new LRUCache(...)
const mySet = (key, value) =>
  cache.set(key, value === undefined ?
        undefinedValue : value)
const myGet = (key, value) => {
  const v = cache.get(key)
  return v === undefinedValue ?
        undefined : v
}
```

Performance

As of January 2022, version 7 of this library is one of the most performant LRU cache implementations in JavaScript.

Benchmarks can be extremely difficult to get right. In particular, the performance of set/get/delete operations on objects will vary *wildly* depending on the type of key used. V8 is highly optimized for objects with keys that are short strings, especially integer numeric strings. Thus any benchmark which tests *solely* using numbers as keys will tend to find that an object-based approach performs the best.

If you put more stuff in the cache, then less recently used items will fall out. That's what an LRU cache is.

For full description of the API and all options, please see [the LRUCache typedocs]

Storage Bounds Safety

This implementation aims to be as flexible as possible, within the limits of safe memory consumption and optimal performance.

At initial object creation, storage is allocated for `max` items. If `max` is set to zero, then some performance is lost, and item count is unbounded. Either `maxSize` or `ttl` *must* be set if `max` is not specified.

If `maxSize` is set, then this creates a safe limit on the maximum storage consumed, but without the performance benefits of pre-allocation. When `maxSize` is set, every item *must* provide a size, either via the `sizeCalculation` method provided to the constructor, or via a `size` or `sizeCalculation` option provided to `cache.set()`. The size of every item *must* be a positive integer.

If neither `max` nor `maxSize` are set, then `ttl` tracking must be enabled. Note that, even when tracking item `ttl`, items are *not* preemptively deleted when they become stale, unless `ttlAutopurge` is enabled. Instead, they are only purged the next time the key is requested. Thus, if `ttlAutopurge`, `max`, and `maxSize` are all not set, then the cache will potentially grow unbounded.

In this case, a warning is printed to standard error. Future versions may require the use of `ttlAutopurge` if `max` and `maxSize` are not specified.

If you truly wish to use a cache that is bound *only* by TTL expiration, consider using a `Map` object, and calling `setTimeout` to delete entries when they expire. It will perform much better than an LRU cache.

Here is an implementation you may use, under the same license as this package:

```
// a storage-unbounded ttl cache that is
        not an lru-cache
const cache = {
  data: new Map(),
  timers: new Map(),
  set: (k, v, ttl) => {
    if (cache.timers.has(k)) {
      clearTimeout(cache.timers.get(k))
    }
    cache.timers.set(
      k,
      setTimeout(() => cache.delete(k),
        ttl),
    )
    cache.data.set(k, v)
  },
  get: k => cache.data.get(k),
  has: k => cache.data.has(k),
  delete: k => {
    if (cache.timers.has(k)) {
      clearTimeout(cache.timers.get(k))
    }
    cache.timers.delete(k)
    return cache.data.delete(k)
  },
  clear: () => {
    cache.data.clear()
    for (const v of
        cache.timers.values()) {
      clearTimeout(v)
    }
    cache.timers.clear()
  },
}
```

If that isn't to your liking, check out @isaacs/ttlcache.

Storing Undefined Values

This cache never stores undefined values, as `undefined` is used internally in a few places to indicate that a key is not in the cache.

You may call `cache.set(key, undefined)`, but this is just an alias for `cache.delete(key)`. Note that this has the effect that `cache.has(key)` will return *false* after setting it to undefined.

```
cache.set(myKey, undefined)
cache.has(myKey) // false!
```