Немного шалим со свежими WeakRef и FinalizationRegistry API

Андрей Печкуров

О докладчике

- Пишу на Java (10+ лет), Node.js (5+ лет)
- Интересы: веб, архитектура, распределенные системы, производительность
- Можно найти тут:
  - https://twitter.com/AndreyPechkurov
  - https://github.com/puzpuzpuz
  - https://medium.com/@apechkurov

- Hazelcast In-Memory Data Grid (IMDG)
- Большой набор распределенных структур данных
- Показательный пример - `Map`, который часто используют как кэш
- Написана на Java, умеет embedded и standalone режимы
- Хорошо масштабируется вертикально и горизонтально
- Часто используется в high-load и low-latency приложениях
- Области применения: IoT, in-memory stream processing, payment processing, fraud detection и т.д.

Hazelcast IMDG Node.js client

- https://github.com/hazelcast/hazelcast-nodejs-client
- Доклад про историю оптимизаций
  - Видео: https://youtu.be/CSnmpbZsVD4
  - Слайды: https://github.com/puzpuzpuz/talks/tree/master/2019-ru-nodejs-library-optimization

План на сегодня

- История вопроса
- Знакомство с WeakRef и FinalizationRegistry API
- Простые примеры использования
- Шалости: Buffer pool для Node.js

История вопроса

Чисто там, где не мусорят

- Первый garbage collector - Lisp, 1959 (лень-матушка)
- Задачи любого GC:
  - Отследить объекты, более недоступные в программе
  - Освободить память в куче под новые объекты
  - Дефрагментировать память (опционально)

Особенности GC

- GC не подразумевает VM, но часто идет в связке
- GC отличаются стратегиями:
  - Tracing (самая популярная)
  - Reference counting
  - Escape analysis (compile-time, стоит особняком)
- Разновидностей конкретных алгоритмов GC - весьма много
- Кроме GC есть такие compile-time штуки, как Automatic Reference Counting (ARC)

Что у нас в JS? (V8)

- Конечно, tracing стратегия
- Комбинирует различные подходы к сборке мусора, чтобы минимизировать stop-the-world паузы
- Кому интересны подробности: https://v8.dev/blog/trash-talk

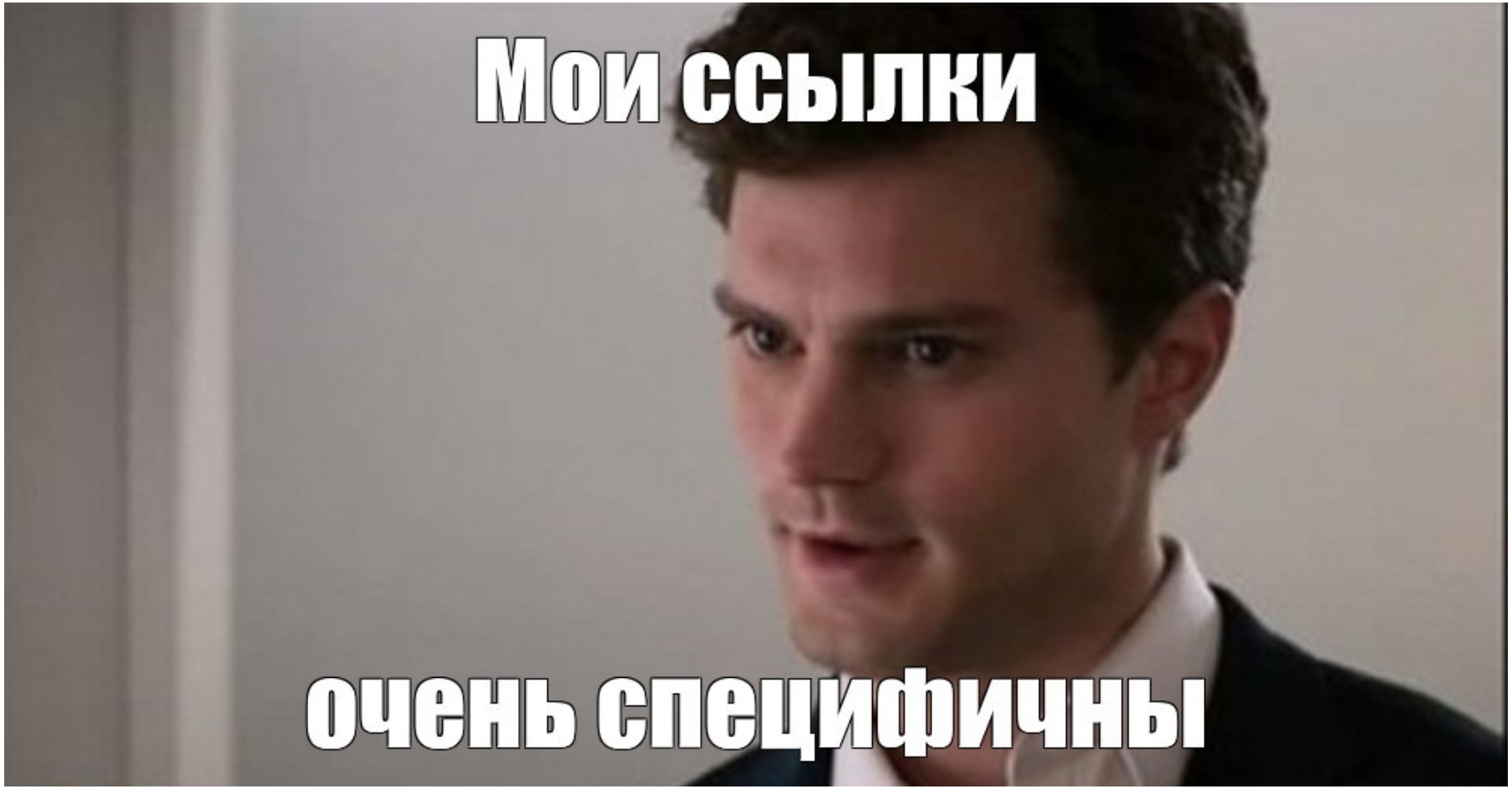
Ссылка есть? А если найду?

- GC оперирует обычными ссылками (strong reference)
- Конечно же, GC учитывает графы зависимостей (и не боится циклов в них)

```
let baz = { answer: 42 };
const foo = { bar: baz };

baz = null;
// a few moments later...
// baz жил, baz жив, baz будет жить
console.log('baz: ', foo.bar);
```

Но что если хочется "необычных" ссылок?



Ссылка есть? А куда дел?

- Во многих языках есть другие виды ссылок
- Например, слабые ссылки (weak reference)
- *Оффтопик*. В ARC слабые ссылки особенно важны

```
let baz = { answer: 42 };
const foo = { bar: new WeakRef(baz) };

baz = null;
// a few moments later...
// baz приказал долго жить
console.log('baz: ', foo.bar.deref());
```

Gotta catch 'em all

- Диалекты Lisp
- Haskell
- Java 1.2, 1998
- Perl 5, 1999
- Python 2.1, 2001
- .NET Framework 1.1, 2002
- И много чего еще

- Знакомство с WeakRef и FinalizationRegistry API

WeakRef

```
// поддерживает только объектные типы
const validRef = new WeakRef({foo: 'bar'});
//const invalidRef = new WeakRef(1); // TypeError

// имеет ровно один метод
const fooBar = validRef.deref();
// в fooBar будет или наш объект, или undefined
if (fooBar !== undefined) {
  console.log('Жив, курилка!');
}
```

WeakRef + WeakMap/WeakSet

- API WeakMap/WeakSet не связаны с WeakRef
- Конечно, WeakRef не препятствует очистке элементов в WeakMap/WeakSet
- Map + WeakRef !== WeakMap (проблема в ссылках между значениями и ключами)
- WeakMap основан на механизме ephemerons (Haskell, Lua), а не на "классических" слабых ссылках

Особенность WeakRef'ов

Фрагмент proposal:

The WeakRefs proposal guarantees that multiple calls to WeakRef.prototype.deref() return the same result within a certain timespan: either all should return `undefined`, or all should return the object.

In HTML, this timespan runs until a microtask checkpoint, where HTML performs a microtask checkpoint when the JavaScript execution stack becomes empty, after all `Promise` reactions have run.

FinalizationRegistry

```
function cleanUp(holdings) {
  for (const i of holdings) {
    console.log(i);
  }
}

const fr = new FinalizationRegistry(cleanUp);
const obj = {};
fr.register(obj, 42);

// после того, как obj собран
// 42
```

КО подсказывает

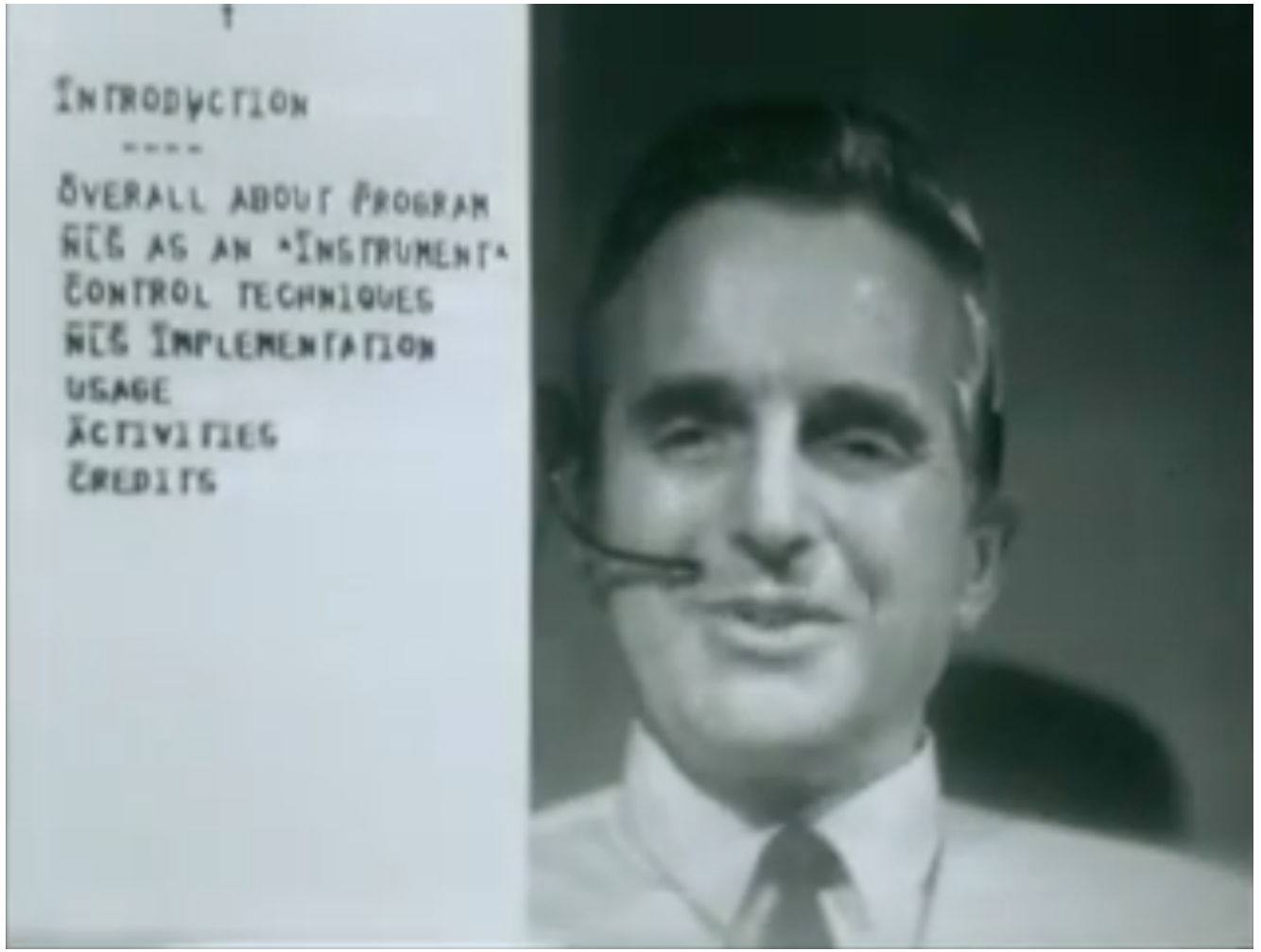
- Время сборки мусора непредсказуемо
- Разные JS движки могут вести себя по-разному
- Финализаторы - нишевая штука, которую стоит избегать в большинстве случаев

Когда ждать?

- Сейчас спецификация на stage 3 в TC39 (предпоследний шаг)
- https://github.com/tc39/proposal-weakrefs
- Можно щупать в Node.js v12+ (и V8) с флагом `--harmony-weak-refs`

Примеры использования

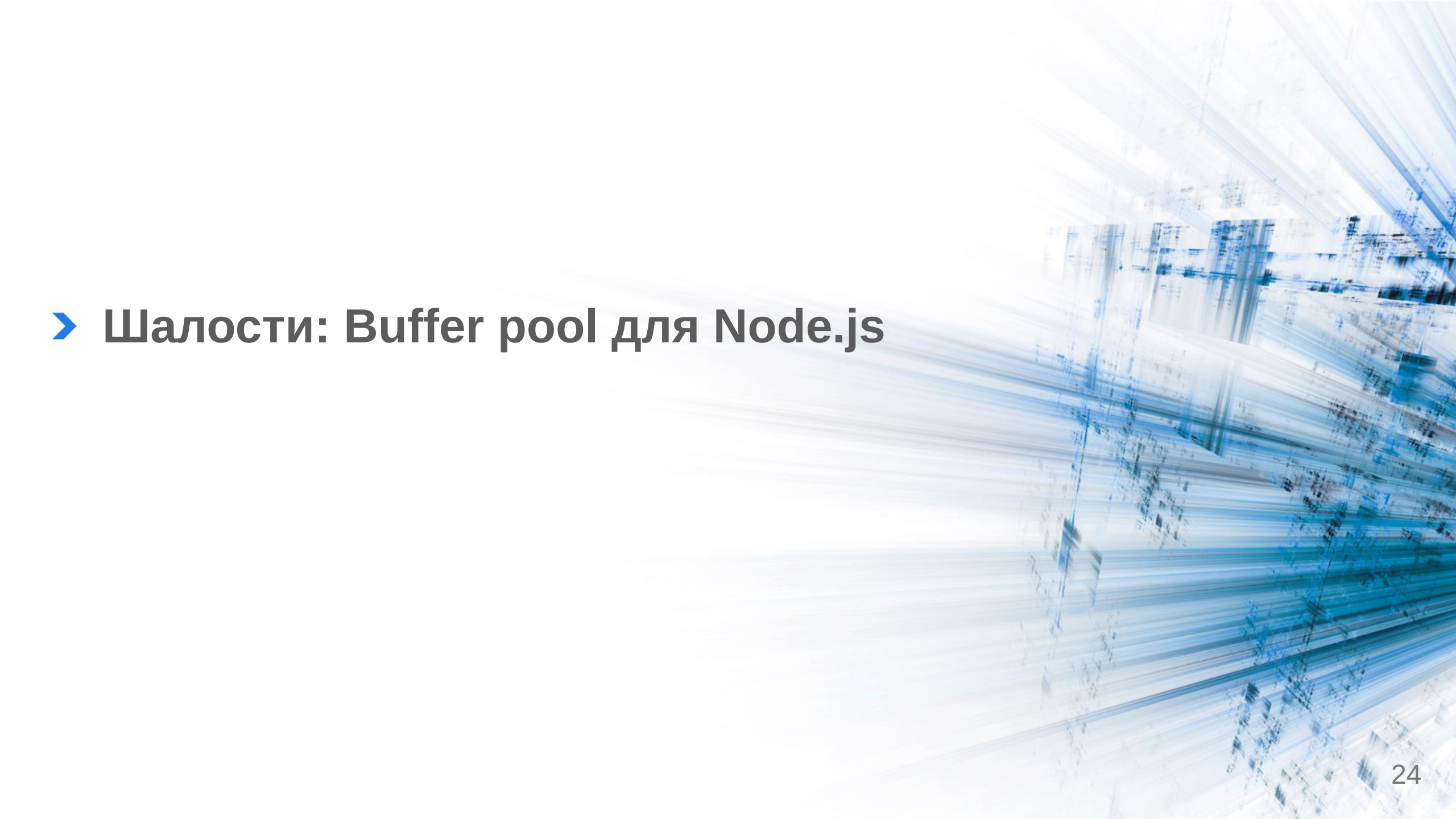
Простые примеры



Менее тривиальные примеры

- Освобождение памяти при работе с объектами, использующими данные на стороне WebAssembly (FR)
- Для обхода проблемы циклических ссылок в модуле `domain` Node.js

Шалости: Buffer pool для Node.js



Buffer API

- https://nodejs.org/api/buffer.html
- Низкоуровневый API для работы с непрерывными массивами байтов
- На heap находится только мета-объект, сами данные - off-heap
- В самом Node.js и многих библиотеках Buffer используются повсеместно

Buffer.allocUnsafe

```
// небезопасно, но быстро :)
let buf = Buffer.allocUnsafe(1024);
// содержимое - произвольное (не ноли)
console.log(buf);
```

Альтернативы Buffer.allocUnsafe

```
// небезопасно и медленно
buf = Buffer.allocUnsafeSlow(1024);
// безопасно и медленно (одни ноли)
buf = Buffer.alloc(1024);
```

Почему Buffer.allocUnsafe быстрее?

Фрагмент из `lib/buffer.js`:

```
Buffer.poolSize = 8 * 1024;

function createPool() {
  poolSize = Buffer.poolSize;
  allocPool = createUnsafeArrayBuffer(poolSize);
  poolOffset = 0;
}
createPool();
```

Почему Buffer.allocUnsafe быстрее?

```
function allocate(size) {
  if (size <= 0) { return new FastBuffer(); }
  if (size < (Buffer.poolSize >>> 1)) {
    if (size > (poolSize - poolOffset)) createPool();
    const b = new FastBuffer(allocPool, poolOffset, size);
    poolOffset += size;
    alignPool();
    return b;
  }
  return createUnsafeBuffer(size);
}

Buffer.allocUnsafe = function allocUnsafe(size) {
  assertSize(size);
  return allocate(size);
};
```

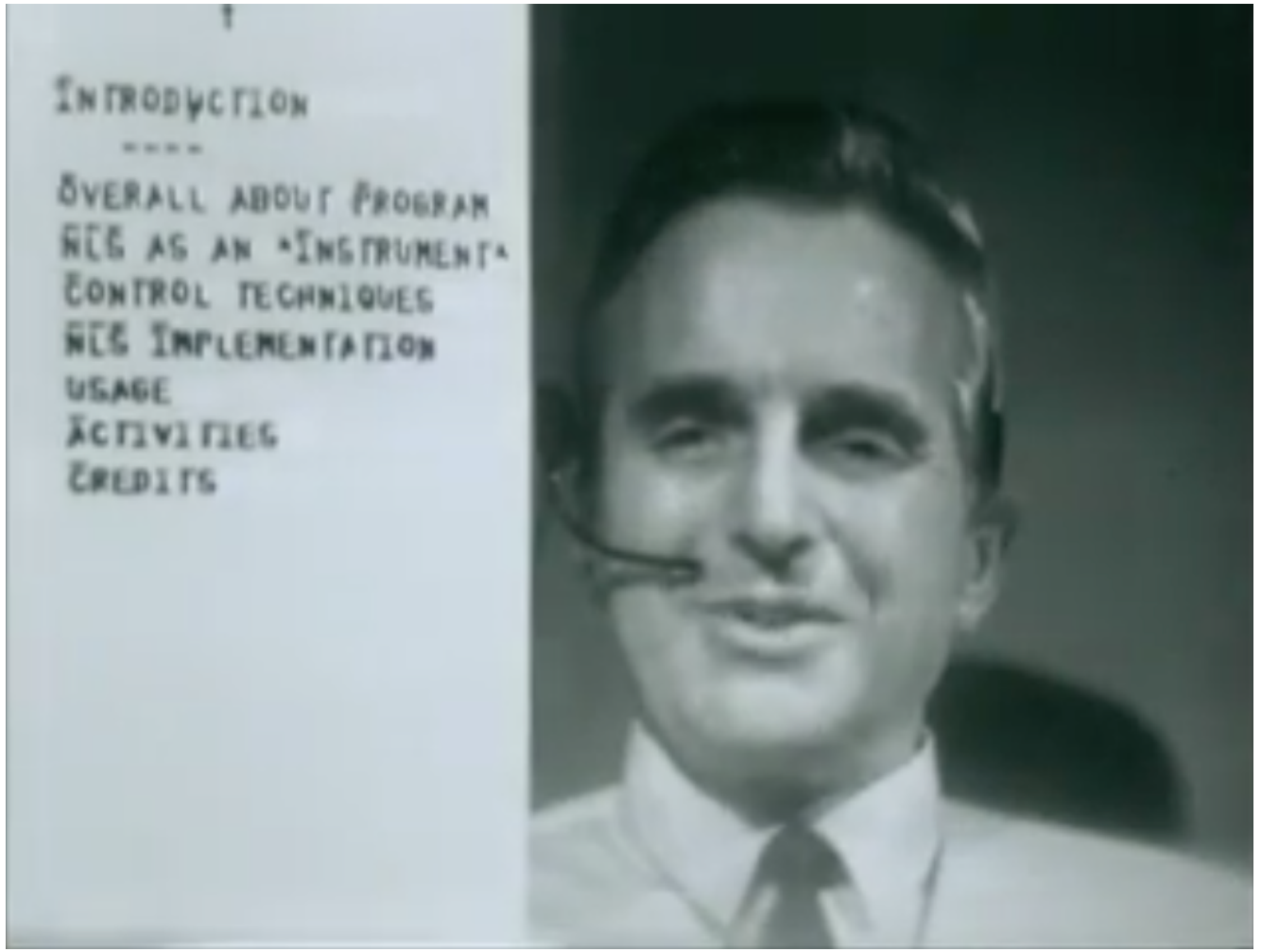
Предпосылки

- От значения `Buffer.poolSize` зависит используется ли "пул", т.е. производительность многих функций из `Buffer`
- Отсюда: https://github.com/nodejs/node/issues/30611
- В ходе обсуждения с контрибьютерами появилась идея
- Почему бы не использовать FinalizationRegistry API для создания "настоящего" пула буферов?
- P.S. В core команде подумывают увеличить дефолт для `Buffer.poolSize`:
  - https://github.com/nodejs/node/issues/27121
  - https://github.com/nodejs/node/pull/30661

Эксперимент с Buffer pool

- Описание эксперимента: https://github.com/nodejs/node/issues/30683
- Ветка с экспериментальной реализацией пула: https://github.com/puzpuzpuz/nbufpool/tree/experiment/fg-api-based-pool

nbufpool


INTRODUCTION
OVERALL ABOUT PROGRAM
NIS AS AN "INSTRUMENT"
CONTROL TECHNIQUES
NIS IMPLEMENTATION
USAGE
ACTIVITIES
CREDITS

Что в итоге?

- Мы (Hazelcast) помешаны на производительности 🙂
- Эксперимент с Buffer pool, скорее всего, продолжится:
  - Manual alloc/free
  - Buddy & slab allocator algorithms
  - Прочие шалости

Call to Action

- Все, кому интересны высокопроизводительные библиотеки (и распределенные системы) - welcome
- https://github.com/hazelcast/hazelcast-nodejs-client
- P.S. Contributions are welcome as well

Спасибо за внимание!

Полезные ссылки

- https://github.com/tc39/proposal-weakrefs
- https://github.com/tc39/proposal-weakrefs/blob/master/history/weakrefs.md
- http://www.cs.bu.edu/techreports/pdf/2005-031-weak-refs.pdf