

# История одной оптимизации производительности Node.js библиотеки

Андрей Печкуров, Hazelcast

# 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>

# О докладе

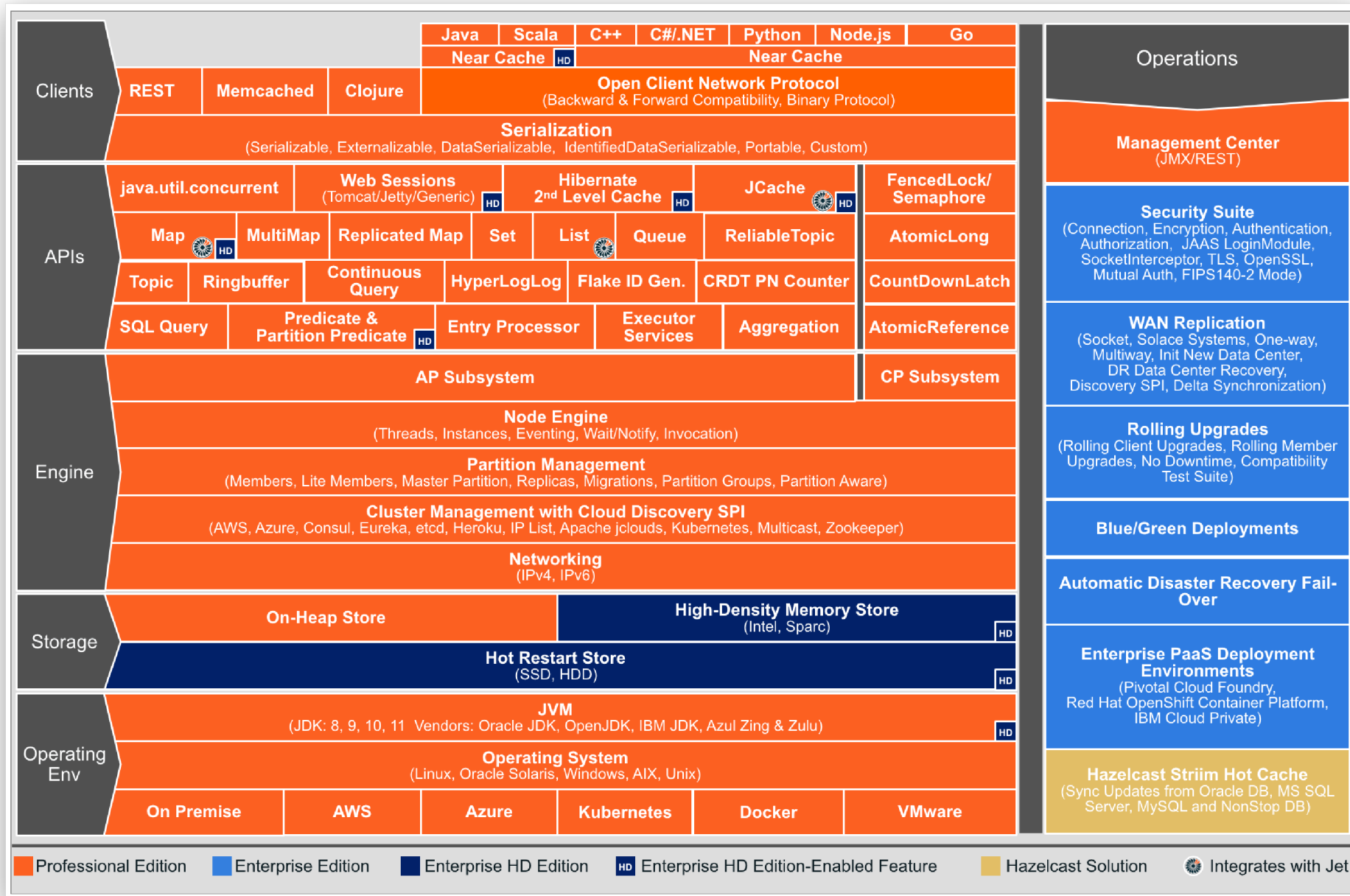
- Тема: оптимизация производительности Node.js библиотек/приложений
- Подопытный: Node.js клиентская библиотека Hazelcast IMDG
- Аудитория: все, кто разрабатывает сетевые приложения на Node.js
- План:
  - i. Знакомство с подопытным
  - ii. Цели и общий подход
  - iii. Бенчмарки и инструменты анализа
  - iv. Оптимизация: гипотезы, эксперименты, результаты
  - v. Планы на будущее

# **1. Знакомство с подопытным**










# Hazelcast IMDG

TODO: описать

# Архитектура Hazelcast IMDG



# Возможности Hazelcast IMDG

|         |   |            |  |  |  |  |  |                 |                          |                |                 |
|---------|---|------------|--|--|--|--|--|-----------------|--------------------------|----------------|-----------------|
| Clients |   |            |  | Java   | Scala  | C++  | C#/.NET  | Python          | Node.js                  | Go             |                 |
|         |   |            |  | Near Cache  |  | Near Cache   |  |                 |                          |                |                 |
|         | REST  | Memcached  | Clojure  | Open Client Network Protocol<br>(Backward & Forward Compatibility, Binary Protocol)            |  |  |  |                 |                          |                |                 |
|         | Serialization<br>(Serializable, Externalizable, DataSerializable, IdentifiedDataSerializable, Portable, Custom)   |            |  |  |  |  |  |                 |                          |                |                 |
| APIs    | java.util.concurrent  |            | Web Sessions<br>(Tomcat/Jetty/Generic)  |  | Hibernate<br>2 <sup>nd</sup> Level Cache  |  | JCache   |                 | FencedLock/<br>Semaphore |                |                 |
|         | Map   | MultiMap   | Replicated Map   |  | Set  | List  | Queue  | ReliableTopic   |                          | AtomicLong     |                 |
|         | Topic   | Ringbuffer | Continuous Query   |  | HyperLogLog  |  | Flake ID Gen.  | CRDT PN Counter |                          | CountDownLatch |                 |
|         | SQL Query   |            | Predicate & Partition Predicate       |  | Entry Processor  |  | Executor Services  |                 | Aggregation              |                | AtomicReference |
|         | AP Subsystem  |            |  |  |  |  |  |                 |                          | CP Subsystem   |                 |

# Hazelcast IMDG Node.js client

- <https://github.com/hazelcast/hazelcast-nodejs-client>
- Node.js 4+
- Стек: TypeScript, promisified API (bluebird)
- Первый стабильный релиз - май 2019



# Особенности библиотеки

- "Умная" клиентская библиотека
- Общается с нодами кластера по [открытому бинарному протоколу](#) поверх TCP
- Поддерживает множество распределенных структур данных

# Пример использования

```
const Client = require('hazelcast-client').Client;  
  
const client = await Client.newHazelcastClient();  
const cache = await client.getMap('my-awesome-cache');  
  
await cache.set('foo', 'bar');  
const cached = await cache.get('foo');
```

## **2. Цели и общий подход**

# Начальные цели

- Анализ текущей производительности перед стабильным релизом
- Включение в релиз "быстрых" правок (при необходимости)
- Постановка планов по дальнейшему анализу и оптимизации
- *Спойлер*: большая часть из этих планов уже реализована

# Оптимизация?



# Оптимизация? Рецепт приготовления

0. Определить метрики производительности и, по возможности, желаемые значения
1. Реализовать бенчмарк
2. Сделать замеры
3. Проблема? Подобрать инструменты анализа
4. Найти бутleneки, выдвинуть гипотезы и провести эксперименты
5. Сделать замеры
6. `goto 0.`

# Возможные метрики

- Сетевая клиентская библиотека
- I/O bound нагрузка
- Основные метрики:
  - Операции в секунду (throughput)
  - Время выполнения операции (~latency)
- Вспомогательные метрики:
  - Загрузка процессора
  - Потребление памяти

# Выбор метрик?

- Оптимизируем throughput
- Желаемые значения: `~\_(\ツ)\_/~`



# Выбор метрик!



### **3. Бенчмарки и инструменты анализа**

# Старый бенчмарк

```
var key = Math.random() * ENTRY_COUNT;
var opType = Math.floor(Math.random() * 100);
if (opType < GET_PERCENTAGE) {
    this.map.get(key).then(this.increment.bind(this));
}
// ...
setImmediate(this.run.bind(this));
```

# Старый бенчмарк: минусы

- Зависимость от `setImmediate()` (macrotask)
- Нет ограничений по кол-ву операций (concurrency limit, backpressure)
- Операции и значения выбираются случайным образом
- Это снижает результаты и детерменистичность

# Новый бенчмарк

```
const benchmark = new Benchmark({  
  nextOp: () => map.get('foo'),  
  totalOpsCount: REQ_COUNT,  
  batchSize: BATCH_SIZE  
});  
await benchmark.run();
```

# Новый бенчмарк: визуализация

TODO: сделать картинку (+ написать про варьирование totalOpsCount и batchSize, а также про perf timers API)

```
op1--->|op6--->| finish  
op2->|op4----->| finish  
op3->|op5-->|op7->| finish
```

# Сценарий

- Приложение-бенчмарк с клиентской библиотекой
- Кластер из одной ноды IMDG (Docker контейнер)
- Локальная машина (loopback address)
- Фиксированные версии Linux, Node.js, IMDG и т.д.
- Операции: `IMap.get()` и `IMap.set()`
- Данные: фиксированные строки с ASCII-символами (3 B, 1 KB, 100 KB)
- Замер: несколько запусков и вычисление среднего результата
- Каждый запуск: 1 млн операций с лимитом 100

# Инструмент #1

- Стандартный профилировщик Node.js
- Основан на V8 sample-based profiler
- Учитывает JS и C++ код
- `node --prof app.js`

- Можно получить человекочитаемое представление:

```
node --prof-process isolate-0xxxxxxxxxxxxx-v8.log > processed.txt
```



# Пример вывода

[Summary]:

| ticks | total | nonlib | name             |
|-------|-------|--------|------------------|
| 4144  | 77.3% | 78.0%  | JavaScript       |
| 1157  | 21.6% | 21.8%  | C++              |
| 374   | 7.0%  | 7.0%   | GC               |
| 51    | 1.0%  |        | Shared libraries |
| 11    | 0.2%  |        | Unaccounted      |

[JavaScript]:

| ticks | total | nonlib | name  |
|-------|-------|--------|---|
| 2104  | 39.2% | 39.6%  | Builtin: StringAdd_CheckNone_NotTenured             |
| 1312  | 24.5% | 24.7%  | LazyCompile: *<anonymous> :1:20                     |
| 484   | 9.0%  | 9.1%   | LazyCompile: *suite.add /home/puzpuzpuz/app.js:68:7 |

...

## Инструмент #2

- Визуализация профиля в виде flame graph
- Действительно помогает обнаруживать ботлнеки
- Отлично работает для event loop'a Node.js
- Спасибо Brendan Gregg, Netflix
- Использовался flamebearer, который дружит с V8 profiler

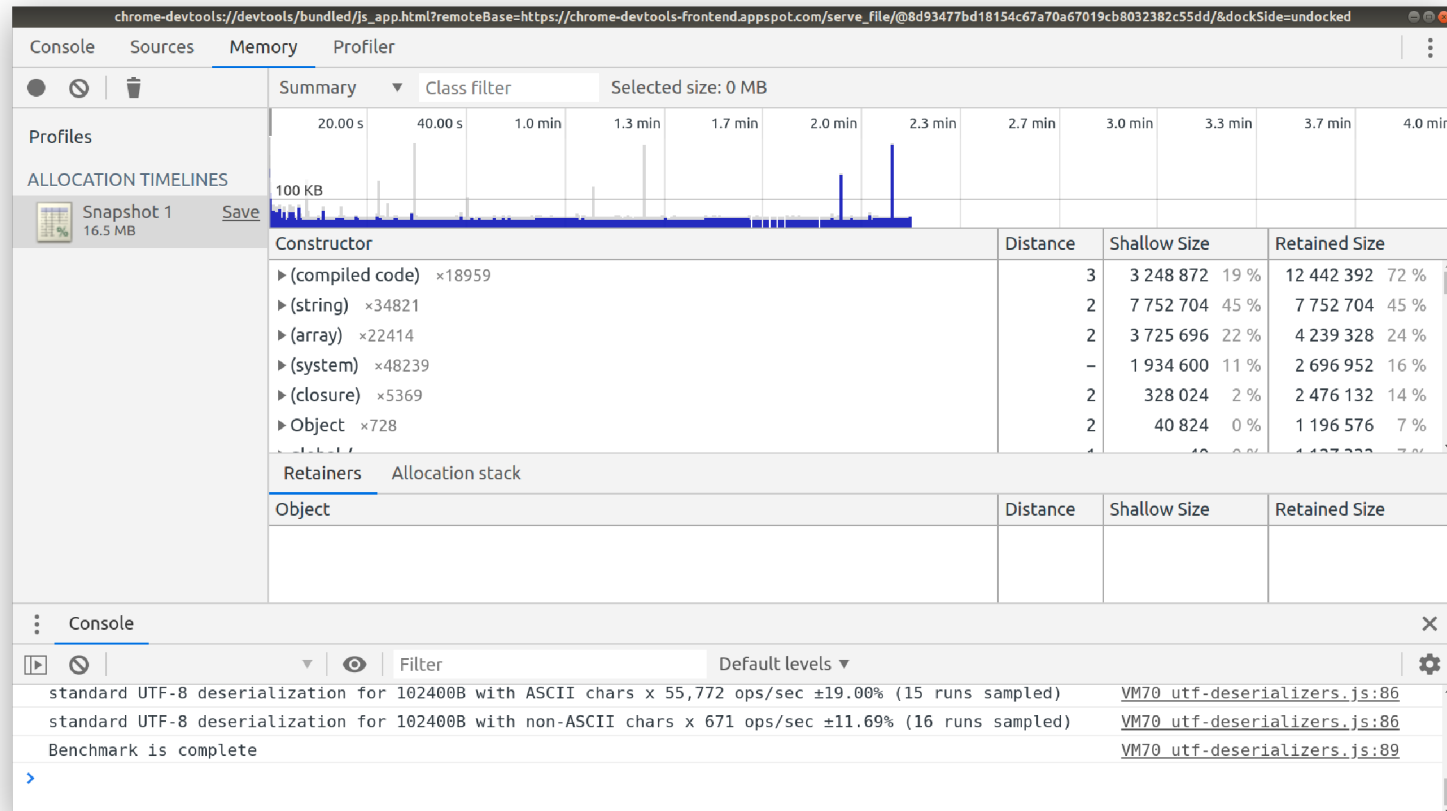
## Пример flame graph

```
$ npm install -g flamebearer  
$ node --prof-process --preprocess -j isolate*.log | flamebearer
```

TODO: добавить картинку

# Инструмент #3

- Профилировщик памяти из Chrome DevTools (Node.js)
- Умеет делать heap snapshot, отслеживать аллокации



## Инструмент #4

- Микробенчмарки для быстрой проверки гипотез
- Использовался фреймворк Benchmark.js (+ node-microtime)

## **4. Оптимизация: гипотезы, эксперименты, результаты**

# Горячий путь

1. Старт операции (создание `Promise` )
2. Сериализация сообщения в бинарный формат
3. Отправка в сеть в `socket.write(...)`
4. Чтение фрейма в `socket.on('data', ...)`
5. Десериализация ответного сообщения
6. Вызов `resolve()` у `Promise` 'а операции

## Базовый замер

| <code>set('foo',<br/>'bar')</code> | <code>set()</code> 1<br>KB | <code>set()</code><br>100 KB | <code>get('foo',<br/>'bar')</code> | <code>get()</code> 1<br>KB | <code>get()</code> 100<br>KB |
|------------------------------------|----------------------------|------------------------------|------------------------------------|----------------------------|------------------------------|
| 76 011                             | 44 324                     | 1 558                        | 90 933                             | 23 591                     | 105                          |



## Видны проблемы?

- С увеличением размера данных производительность падает линейно
- Java клиент в сценарии для `get( 'foo', 'bar' )` быстрее примерно в 5 раз (сравнение заведомо некорректное)

TODO: тут будет еще куча слайдов

## Полезные ссылки

- <https://hazelcast.org/>
- <https://github.com/hazelcast/hazelcast-nodejs-client>
- <https://nodejs.org/en/docs/guides/simple-profiling/>
- <https://nodejs.org/en/docs/guides/dont-block-the-event-loop/>
- <https://blog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810>

**Спасибо за внимание!**

**Время для Q&A**