

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: подход к оптимизации производительности Node.js библиотек
- Подопытный: клиентская Node.js библиотека Hazelcast IMDG
- Аудитория: все, кто разрабатывает сетевые приложения на Node.js
- План:
 - #1: Знакомство с подопытным
 - #2: Цели и общий подход
 - #3: Бенчмарки и инструменты анализа
 - #4: Оптимизация: замеры, гипотезы, эксперименты
 - #5: Планы на будущее

#1: Знакомство с подопытным

Hazelcast IMDG

- <https://hazelcast.org/>
- Hazelcast In-Memory Data Grid (IMDG)
- Большой набор распределенных структур данных (AP и CP согласно CAP теореме)
- Написана на Java, умеет embedded и standalone режимы
- Хорошо масштабируется вертикально и горизонтально
- Часто используется в high-load и low-latency приложениях

Возможности Hazelcast IMDG

Clients				Java	Scala	C++	C#/.NET	Python	Node.js	Go	
				Near Cache 		Near Cache					
	REST	Memcached	Clojure	Open Client Network Protocol (Backward & Forward Compatibility, Binary Protocol)							
	Serialization (Serializable, Externalizable, DataSerializable, IdentifiedDataSerializable, Portable, Custom)										
APIs	java.util.concurrent		Web Sessions (Tomcat/Jetty/Generic) 		Hibernate 2 nd Level Cache 		JCache 		FencedLock/ 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
- Умеет near cache, retry on failure, client stats и многое другое

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');  
console.log(cached); // bar
```

#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:* на сегодня большая часть из этих планов уже реализована

Оптимизация?



Оптимизация? Рецепт приготовления

0. Определить метрики производительности (+ желаемые значения)
1. Реализовать бенчмарк
2. Сделать замеры производительности
3. Проблема? Подобрать инструменты анализа
4. Найти узкие места, 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 (условно, latency)
- Вспомогательные метрики:
 - Загрузка процессора
 - Потребление памяти

Выбор метрик?

- Оптимизируем throughput
- Желаемые значения: $\neg(\text{ツ})\neg$

Выбор метрик!



#3: Бенчмарки и инструменты анализа

Старый бенчмарк

```
// ...
run: function () {
  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

- Все операции стартуют через рекурсивный `setImmediate()`
- Нет ограничений по количеству операций (concurrency limit, backpressure)
- Операции и входные данные выбираются случайным образом
- Все это снижает результат и ухудшает детерминированность

Новый бенчмарк

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

Новый бенчмарк: плюсы

- Операции стартуют параллельно
- Общее число одновременно стартованных операций ограничено
- Операции и входные данные predetermined

Новый бенчмарк: визуализация

Пример с `batchSize = 3` и `totalOpsCount = 7`:

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

Простой `Promise.all()`:

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

Сценарий бенчмарка

- 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-0xxxxxxxxxxxxxxxx-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 ./app.js:68:7
...			
8	0.1%	0.2%	LazyCompile: ~<anonymous> ./util.js:51:44

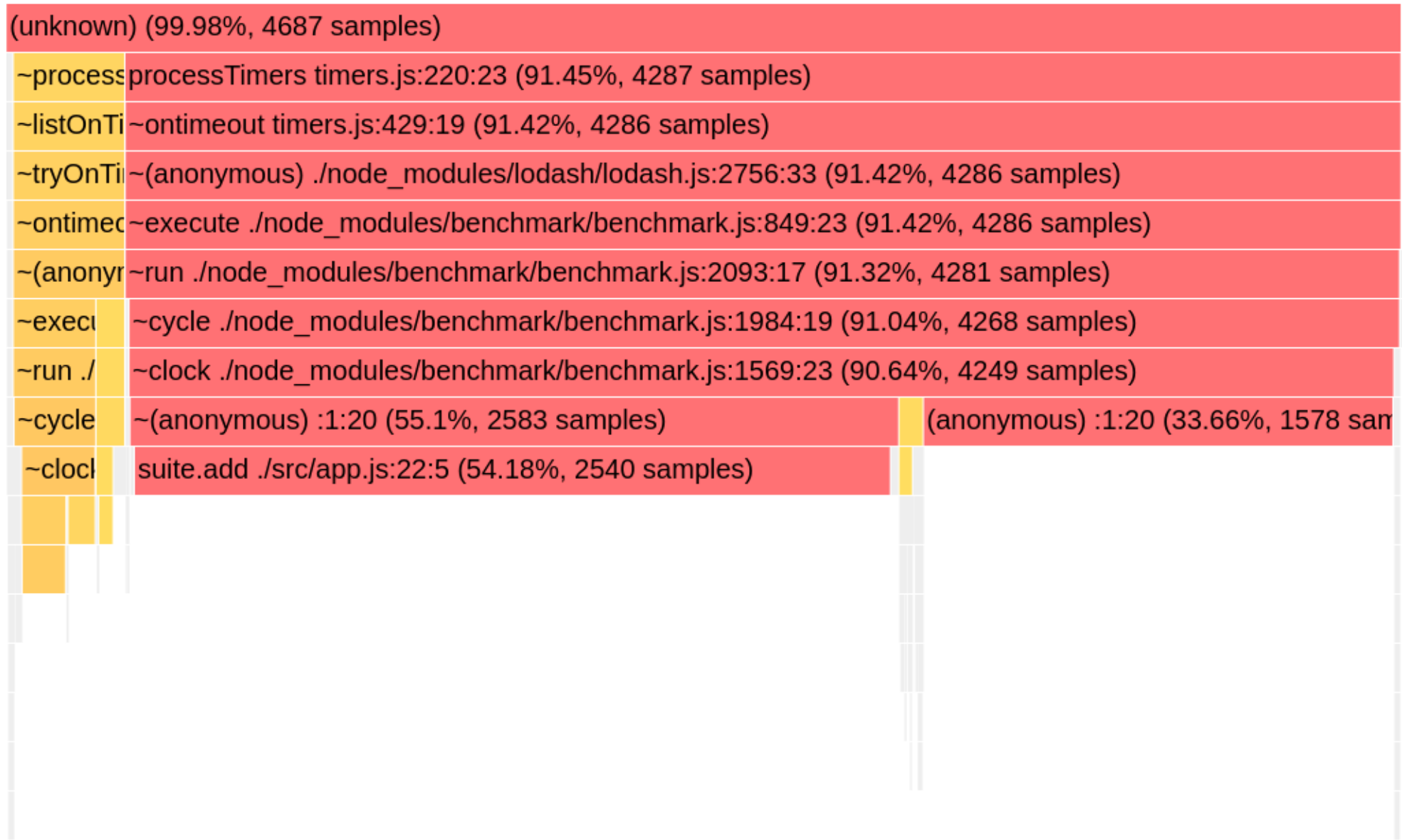
...

Инструмент #2

- Визуализация профиля в виде flame graph
- Действительно помогает обнаруживать ботлнеки
- Отлично работает для event loop'a Node.js
- Спасибо Brendan Gregg, Netflix, [придумавшему подход](#) в 2013
- Наиболее популярный инструмент - [Ox](#) (V8, perf, DTrace)
- Мы использовали [flamebearer](#) (V8)

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

Пример простейшего flame graph

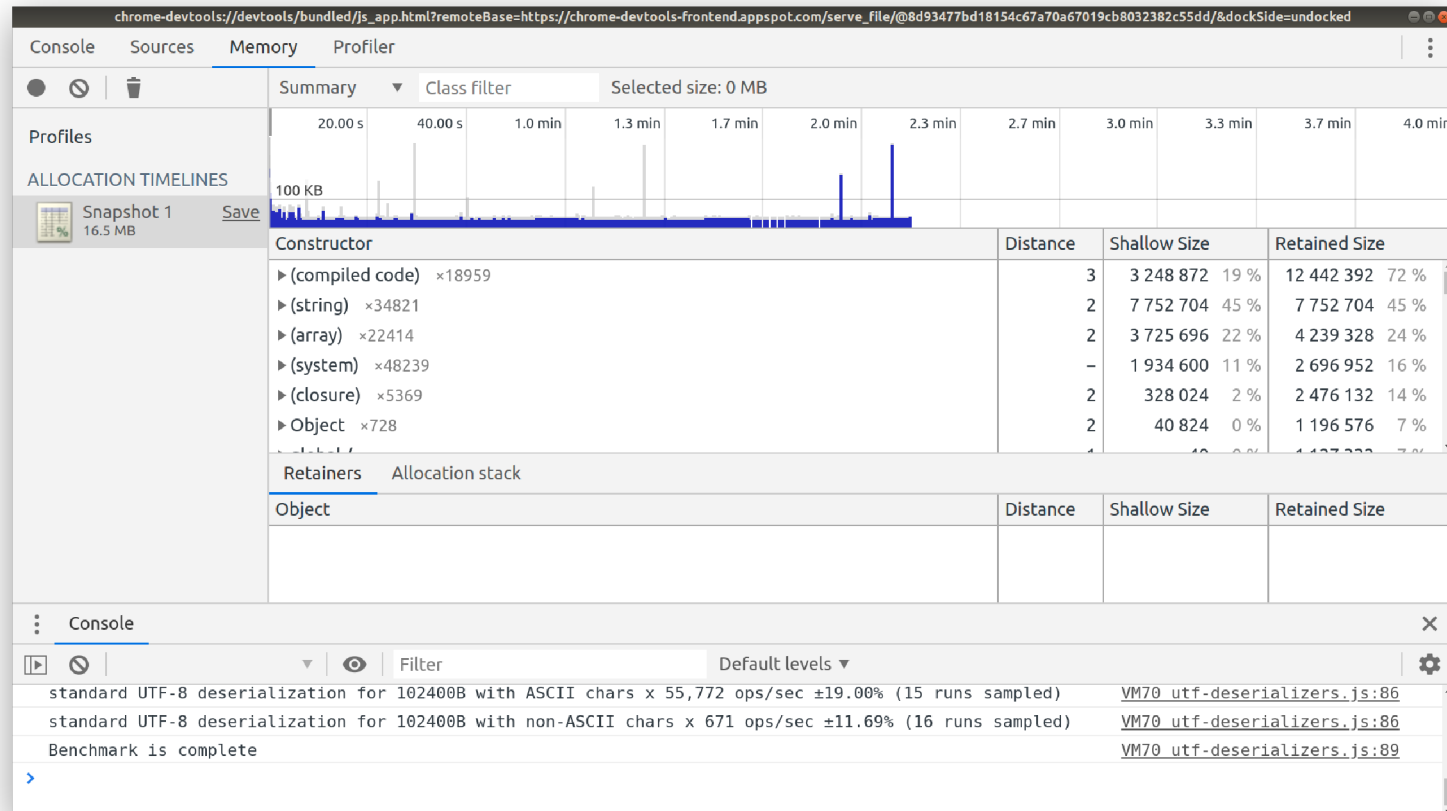


Пример flame graph из реального мира



Инструмент #3

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



Инструмент #4

- Микробенчмарки для быстрой проверки гипотез
- Использовался фреймворк [Benchmark.js](#) (+ node-microtime)
- *Предупреждение*: могут показывать температуру в Антарктиде

Инструмент #5

- Proof of concept (PoC)
- Все средства хороши, но нужен весь функционал кода на горячем пути
- *П.С.:* это не совсем инструмент, но не упомянуть нельзя

Проверяем чеклист

- ☒ Метрики
- ☒ Бенчмарк
- ☒ Инструменты анализа
- ☐ Оптимизация

#4: Оптимизация: замеры, гипотезы, эксперименты

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

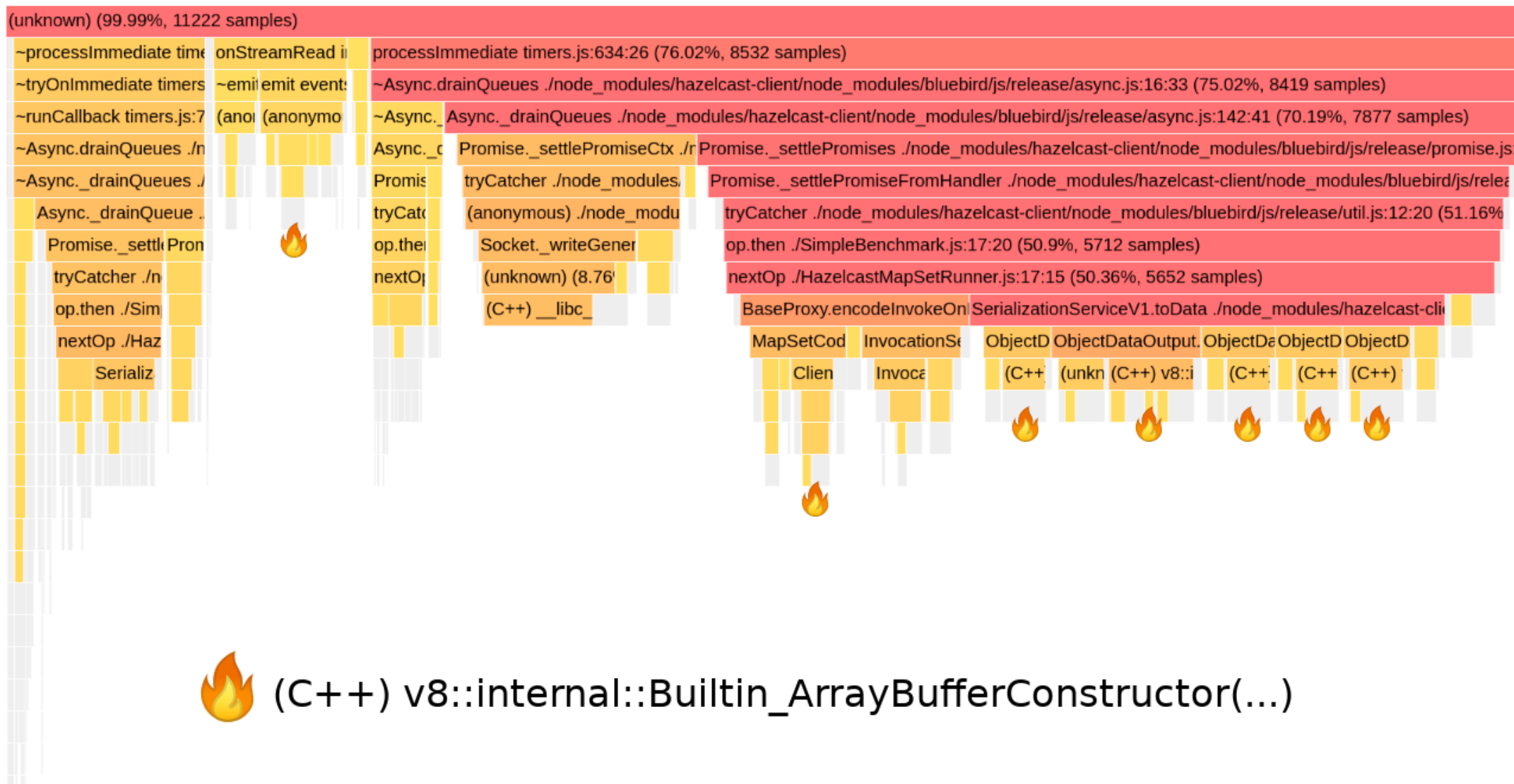
Базовый замер

	3 В	1 KB	100 KB
Map#get()	90 933	23 591	105
Map#set()	76 011	44 324	1 558

Видны проблемы?

- Java-клиент для `get('foo', 'bar')` быстрее примерно в 5 раз (сравнение заведомо некорректное)
- Производительность практически линейно зависит от размера данных

Профилировщик, приди! (запись 3 В)



Хьюстон, у нас аллокации

- Для работы с бинарными данными, конечно, используется [Buffer](#)
- В на горячем пути много `Buffer#alloc()/#allocUnsafe()` , а это "дорогая" операция
- Во время сериализации одной операции происходит несколько аллокаций, а затем буферы копируются в финальный
- Это упрощает код, но производительность страдает
- Сначала делаем PoC с полумерой, поскольку полная правка требует много времени

РoC с полумерой

```
export class ObjectDataOutput implements DataOutput {  
  
    protected buffer: Buffer;  
    private pos: number;  
  
    constructor() {  
        // пробуем аллоцировать жадно  
-        this.buffer = Buffer.allocUnsafe(1);  
+        this.buffer = Buffer.allocUnsafe(1024);  
  
        // ...  
    }  
}
```

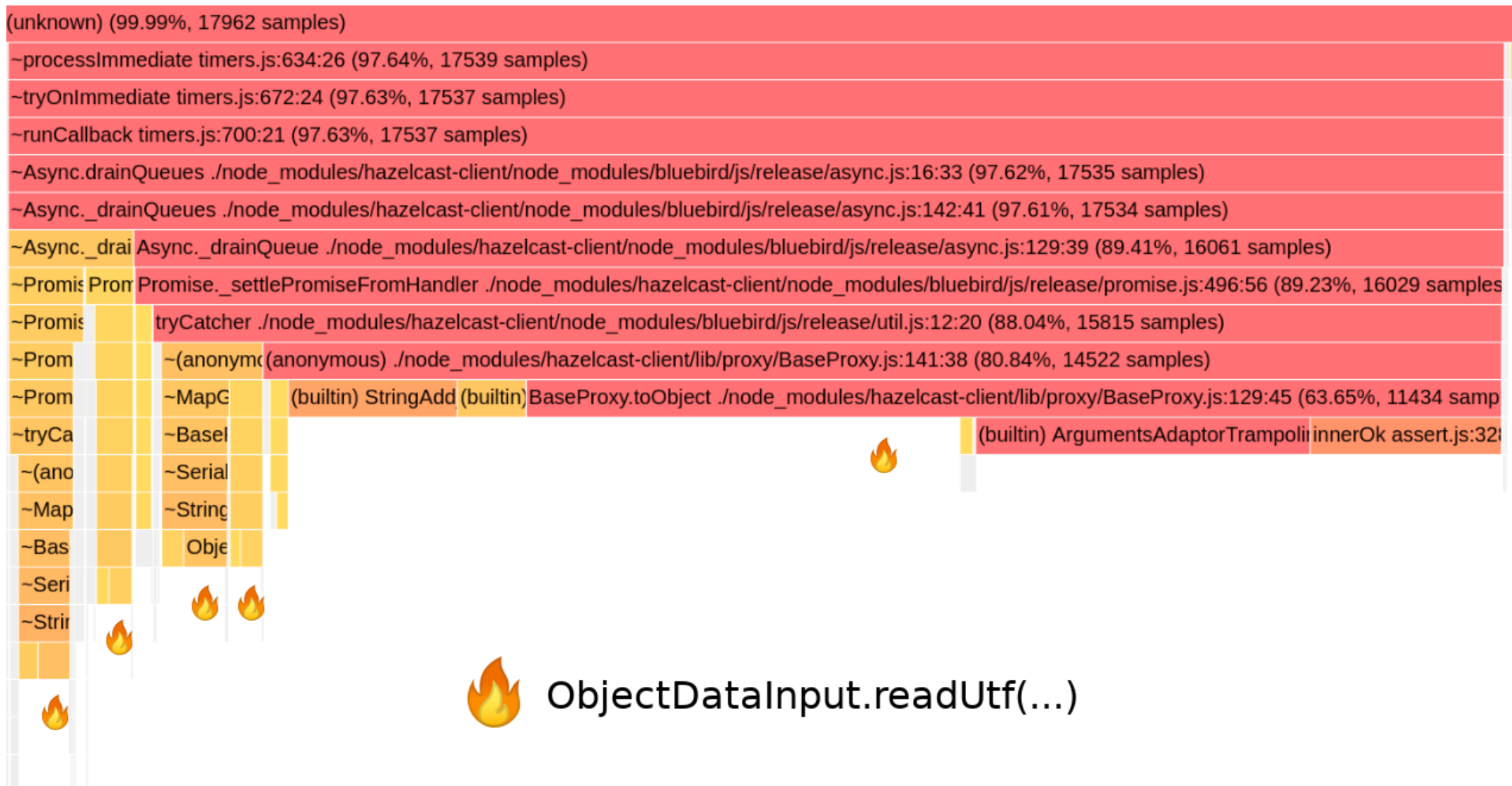
Замер производительности PoC

	get () 3 B	get () 1 KB	get () 100 KB	set () 3 B	set () 1 KB	set () 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
PoC	104 854	24 929	109	95 165	52 809	1 581
	+15%	+5%	+3%	+25%	+19%	+1%

Промежуточные итоги

- Гипотеза верна и правка идет в ближайший релиз
- Нужно избавиться от оставшихся лишних аллокаций в будущих релизах
- Результаты для больших размеров оставляют желать лучшего
- Так что же у нас с зависимостью от размера данных?

Профилировщик, приди! (чтения 100 KB)



А что это у нас там?

```
private readUTF(pos?: number): string {  
    const len = this.readInt(pos);  
    // ...  
    for (let i = 0; i < len; i++) {  
        let charCode: number;  
        leadingByte = this.readByte(readingIndex) & MASK_1BYTE;  
        readingIndex = this.addOrUndefined(readingIndex, 1);  
        const b = leadingByte & 0xFF;  
        switch (b >> 4) {  
            // ...  
        }  
        result += String.fromCharCode(charCode);  
    }  
    return result;  
}
```

Предварительная оптимизация?

- Итак, у нас нестандартная (де)сериализация UTF-8 строк
- Похоже на предварительную оптимизацию
- Почему бы не сравнить со стандартным API?

```
// сериализация
buf.write(inStr, start, end, 'utf8');
// десериализация
const outStr = buf.toString('utf8', start, end);
```

Микробенчмарк

	100 B ASCII	100 KB ASCII	100 B UTF	100 KB UTF
custom	1 515 803	616	1 093 390	613
standard	11 297 821	68 721	1 311 610	794
	+645%	+11 056%	+20%	+29%

* Результаты для десериализации в ops/sec

Проваливаемся в кроличью нору

- `Buffer#toString()`
- `node:buffer.js#stringSlice()`
- `node:node_buffer.cc#StringSlice()`
- `node:StringBytes#Encode()`
- `v8:String#NewFromUtf8()`
- `v8:Factory#NewStringFromUtf8()`
- `v8:Factory#NewStringFromOneByte()`

Что там, в hope?

```
// v8:Factory#NewStringFromUtf8()  
MaybeHandle<String> Factory::NewStringFromUtf8(  
    Vector<const char> string,  
    PretenureFlag pretenure  
) {  
    // Check for ASCII first since this is the common case.  
    const char* ascii_data = string.start();  
    int length = string.length();  
    int non_ascii_start = String::NonAsciiStart(ascii_data, length);  
    if (non_ascii_start >= length) {  
        // If the string is ASCII, we do not need to convert  
        // the characters since UTF8 is backwards compatible with ASCII.  
        return  
            NewStringFromOneByte(  
                Vector<const uint8_t>::cast(string), pretenure);  
    }  
    // ...
```

РoС для сериализации

	get () 3 B	get () 1 KB	get () 100 KB	set () 3 B	set () 1 KB	set () 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
PoC	122 458	104 090	7 052	110 083	73 618	8 428
	+34%	+341%	+6 616%	+45%	+66%	+440%

Промежуточные итоги

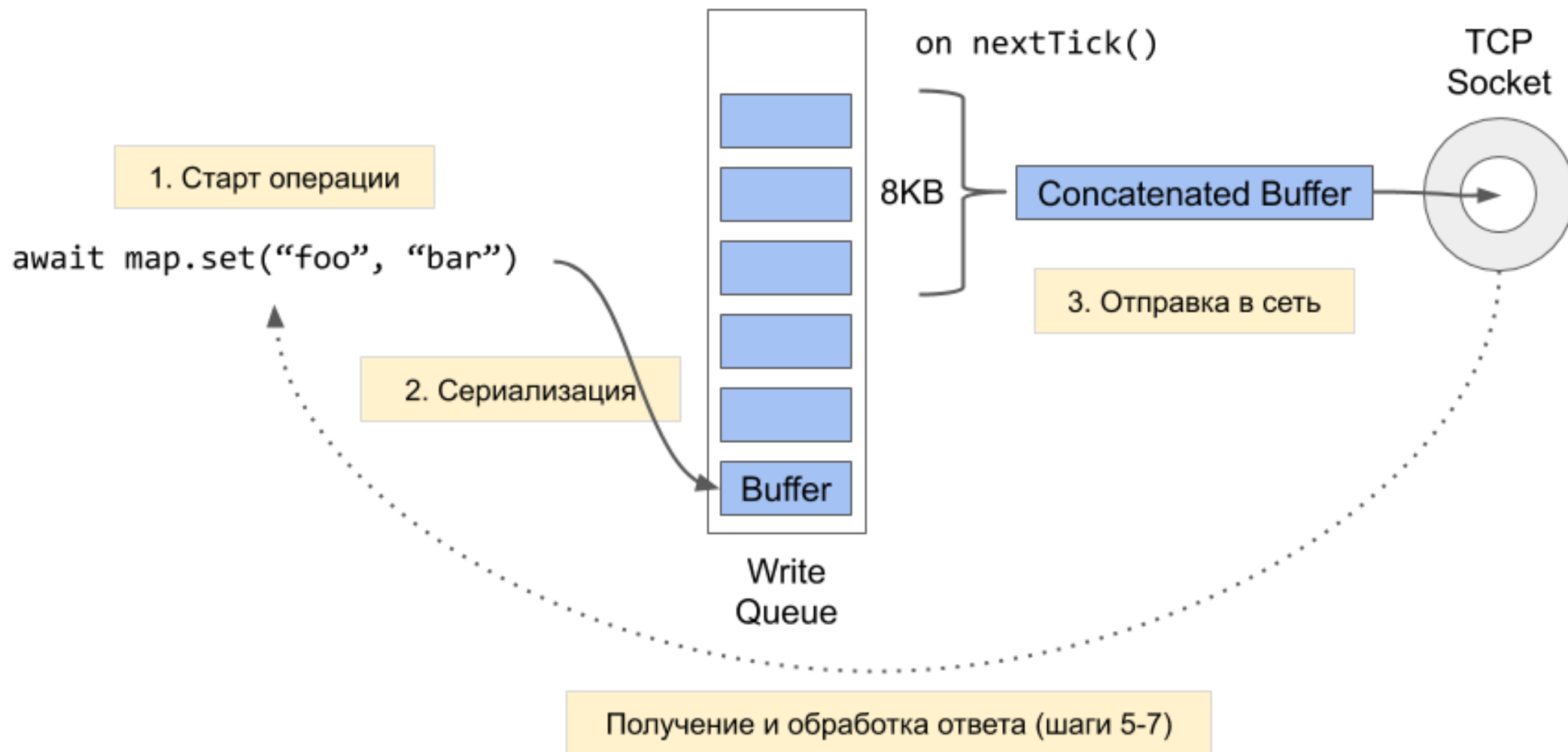
- Гипотеза верна и правка идет в ближайший релиз

Первый публичный релиз 🎉

	<code>get()</code> 3 B	<code>get()</code> 1 KB	<code>get()</code> 100 KB	<code>set()</code> 3 B	<code>set()</code> 1 KB	<code>set()</code> 100 KB
v0.10.0	90 933	23 591	105	76 011	44 324	1 558
v3.12	132 855	120 670	8 756	127 291	94 625	10 617
	+46%	+411%	+8 239%	+67%	+113%	+581%

TODO: тут будет еще куча слайдов

Логика работы Automated Pipelining



#5: Планы на будущее

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!

Особенность `Buffer#allocUnsafe()`

- TODO рассказать про встроенный пул

Еще раз спасибо за внимание!

Время для Q&A