KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

12/2014

Jahrgang 7

Editorial

Wunschvorschlag

Jetzt, da Weihnachten kurz vor der Tür steht, wird sich doch sicher der eine oder andere einen Rechner gewünscht haben. Vielleicht sogar einen *Arduino* oder *Raspberry Pi*, den er für die Haussteuerung einzusetzen gedenkt. Während der eine damit nur die Beleuchtung steuern will, mag der andere vielleicht die Heizung damit regeln. Auf jeden Fall kann so aktiv ein kleiner Beitrag dazu geleistet werden, die (eigene) Welt ein wenig besser zu gestalten.

Wem das noch nicht genug ist, der kann übrigens noch mehr für die Umwelt tun. Beispielsweise könnte er seine aktuelle Heizung gegen ein Server-Rack eintauschen.

Dazu wendet man sich an den *Cloud*-Dienstleister seines Vertrauens, schafft ein bisschen Platz im Keller und erlaubt jenem, einen Rechnerschrank aufzustellen. Die beim Rechnen anfallende Wärme wird dann dazu genutzt, um einen 500 bis 2000 Liter fassenden Warmwasserspeicher aufzuheizen. Mit diesem kann der Hausherr dann sein Haus beheizen und mit warmen Wasser versorgen.

Voraussetzung dafür ist nur ein Drehstromanschluss und Internet mit 50 MBit/s oder mehr. Ersteres ist in vielen Häusern schon vorhanden und von zweitem kann man ja in vielen Gegenden schon profitieren. Der Strom wird vom Dienstleister getrennt abgerechnet. Allerdings muss man für die Installation 12 000 € berappen. Das muss also jeder für sich durchrechnen und mit dem Einbau einer normalen Heizung vergleichen. Aber bei oberflächlicher Betrachtung dürften die laufenden Heizkosten gleich Null sein. Das könnte sich also lohnen.

Die Idee ist gar nicht mal so schlecht. Während der Hausbesitzer von der Wärme profitiert, erreicht der Dienstleister eine Verteilung, die man sich sonst nur wünschen kann. Wie oft aber ein Techniker vorbeischauen muss, ließ sich auf die Schnelle nicht herausbekommen. Kommt der nur zu bestimmten Zeiten? Oder guckt er auch schonmal nachts um zwei Uhr vorbei, wenn Not am Mann ist?

Wie dem auch sei, in diesem Jahr ist es dafür wohl zu spät. Nächstes Jahr könnte es damit noch klappen. Und vielleicht gibt es ja dann noch freie Rechenzeit als Schmankerl obendrein.

Wir wünschen insbesondere allen Lesern schöne Feiertage und ein gesundes und erfolgreiches neues Jahr.

Ihr Michael Wiedeking

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 7, Nummer 12, Dezember 2014
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Artikel

Kolumnen

Phantome im Browser

von Frank Goraus

Tests von Java-Klassen sind dank *Unit*-Tests relativ einfach. Doch was wenn man eine Web-Seite oder eine ganze Web-Anwendung automatisiert testen möchte? Es sind inzwischen viele Lösungen für dieses Problem auf dem Markt. Doch die wenigsten sind wirklich leichtgewichtig oder funktionieren wie ein echter Browser, was Tests von *JavaScript*- und *AJAX*-lastigen Seiten schwierig gestaltet. *PhantomJS* bietet eine Lösung für beides und noch viel mehr.

Auf Safari im Programmierdschungel

von Marc Spanagel

JavaScript ist auf dem Vormarsch. Seit Java 8 ist es nun auch fester Bestandteil des JDK und hört auf den deutschen Namen *Nashorn*. Dynamisches JavaScript auf der JVM – wie funktioniert das denn?

Ausbruchssicher

von Michael Wiedeking

Versucht man auf ein *Array*-Element zuzugreifen, das nicht existiert, so wird dies in nicht ganz so toleranten Sprachen zurecht mit etwas wie einer *IndexOutOfBounds*-Ausnahme geahndet. Gelegentlich würde es aber schon genügen, wenn auch noch beim Verlassen des gültigen Bereichs ein Element geliefert würde.

Phantome im Browser

Ein Einblick in PhantomJS

von Frank Goraus

Tests von Java-Klassen sind dank *Unit*-Tests relativ einfach. Doch was wenn man eine Web-Seite oder eine ganze Web-Anwendung automatisiert testen möchte? Es sind inzwischen viele Lösungen für dieses Problem auf dem Markt. Doch die wenigsten sind wirklich leichtgewichtig oder funktionieren wie ein echter Browser, was Tests von *JavaScript*- und *AJAX*-lastigen Seiten schwierig gestaltet. *PhantomJS* bietet eine Lösung für beides und noch viel mehr.

Was ist PhantomJS?

PhantomJS bezeichnet sich selbst als „headless WebKit scriptable". Was soviel heißen soll, dass sich darin ein Browser auf Basis der *WebKit-Engine* versteckt, wie er in Google Chrome und Safari zum Einsatz kommt. Dieser ist jedoch „kopflos", das heißt ohne GUI, und kann per Skript gesteuert werden. Dadurch bietet es sich natürlich perfekt für automatisierte Tests an, welche so auch auf Server-Systemen ohne grafische Oberfläche eingesetzt werden können. Darüber hinaus kann es auch für Automatisierungs-Skripte auf Web-Seiten oder Performanzanalysen im Netzwerk genutzt werden. Und falls es notwendig sein sollte einzelne Schritte zu visualisieren, kann man sehr bequem jederzeit Screenshots, der (nicht) zu sehenden Seiten machen.

Das klingt zwar nicht nach Unmengen an Features, aber damit kann man trotzdem ziemlich viel anstellen. Weiterhin besitzt *PhanotmJS* ein großes Ecosystem und wird auch in anderen Projekten eingebunden. Es ist also nicht verwunderlich, dass große namhafte Projekte wie *Bootstrap, jQuery Mobile*, *Ember.js*, *Less.js* oder *Modernizer* auch *PhantomJS* zum Testen einsetzen.

Erste Schritte

Um *PhantomJS* einzusetzen, muss man zunächst entweder das *Binary* (gibt es für Windows, Linux und Mac) von der Projektseite laden oder man verwendet den *Node Package Manager* und installiert es mit folgendem Befehl auf seinem System.

```
npm install -g phantomjs
```

Nach der Installation (und eventuell Einbindung im Pfad) steht einem das Kommandozeilen-Tool *phantomjs* zur Verfügung. Startet man dieses, wird man mit einer neuen Konsolenansicht begrüßt und kann zum Beispiel die Version abfragen:

```
phantom.version
```

Oder sich Informationen über den zugrundeliegenden Browser geben lassen:

```
window.navigator
```

Bei letzterem Aufruf wird eines gewahr, nämlich dass durch das Starten der *PhantomJS*-Konsole bereits der interne Browser im Hintergrund werkelt (siehe Abbildung 1).

Eigene Skripte

PhantomJS wäre jedoch ziemlich unkomfortabel, wenn man alles über die Konsole machen müsste. Und so gibt es die Möglichkeit vorgefertigte Skripte zu erstellen, die man als Übergabeparameter beim Starten des Tools angibt, und die dann direkt ausgeführt werden.

Als einfaches Beispiel soll folgendes Skript dienen, das als *script.js* abgespeichert wird und dann über

```
phantomjs script.js
```

gestartet wird:

```
//Lädt das Webpage-Modul von PhantomJS
var webpage = require('webpage');

//Erzeugt eine leere Seite (Tab)
var page = webpage.create();

//Test-URL
var url = "http://www.mathema.de";

//Lädt die Seite, und speichert diese
//bei Erfolg als Screenshot ab
```

```
E:\>phantomjs
phantomjs> phantom.version
{
    "major": 1,
    "minor": 9,
    "patch": 8
}
phantomjs> window.navigator
{
    "appCodeName": "Mozilla",
    "appName": "Netscape",
    "appVersion": "5.0 (Windows NT 6.1; WOW64) AppleWebKit/534.34 (KHTML, like Ge
cko) PhantomJS/1.9.8 Safari/534.34",
    "cookieEnabled": true,
    "language": "de-DE",
    "mimeTypes": {
        "length": 0
    },
    "onLine": false,
    "platform": "Win32",
    "plugins": {
        "length": 0
    },
    "product": "Gecko",
    "productSub": "20030107",
    "userAgent": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/534.34 (KHTML,
like Gecko) PhantomJS/1.9.8 Safari/534.34",
    "vendor": "Apple Computer, Inc.",
    "vendorSub": ""
}
phantomjs> _
```

Abbildung 1

```
page.open(url, function(status) {

  if (status !== 'success') {
    console.log("Fehler: "+status);
    phantom.exit();
  }

  //Screenshot-Erstellung
  page.render('test.png');

  //Exit nicht vergessen!
  phantom.exit();
});
```

Zu Beginn muss erst einmal das *Webpage-Modul* [1] geladen werden, womit quasi der *PhantomJS*-interne Browser gestartet wird. Danach erzeugt man damit eine neue Seite, was dem Öffnen eines leeren Tabs entspricht. In dieser Seite wird dann die gewünschte URL geöffnet. Beim Aufruf der *open*-Methode muss man zusätzlich eine *Callback*-Funktion angeben, die aufgerufen wird sobald das Öffnen erfolgreich oder erfolglos war. Welches von beiden erfährt man über den Übergabeparameter, der an den Callback übergeben wird. Sollte dieser nicht *success* enthalten, ist irgendein Fehler aufgetreten.

Andernfalls kann man dann auf der geladenen Seite arbeiten. In unserem Fall machen wir nichts weiter als die *render*-Methode aufzurufen, mit der man einen Screenshot der Seite erstellen kann und diesen als Bild (*png*, *jpeg*, *gif*) oder PDF abspeichert. Per *Default* wird das Format an der Dateiendung erkannt, man kann dies jedoch auch selbst definieren:

```
page.render('screenshot.'+Date.now(),
   {format: 'jpeg', quality: '85'} );
```

Hier wird als Dateiendung der aktuelle *Timestamp* verwendet und über ein *Options*-Objekt angegeben, dass das Bildformat ein JPEG sein soll, sowie dessen Komprimierung nur 85% der Bildqualität entsprechen soll.

Ganz wichtig ist, dann am Ende der Verarbeitung, die *PhantomJS*-Umgebung anzuweisen, dass sie sich beenden soll. Andernfalls läuft das Kommandozeilen-Tool weiter, obwohl die Verarbeitung bereits vollständig abgeschlossen wurde!

Möchte man statt eines Screenshots lieber den Inhalt der Seite haben, so besitzt das *page*-Objekt zwei *Properties*:

```
page.plainText
page.content
```

In der Ersten befindet sich die Seite als reine Text-Repräsentation, was bedeutet, dass alle *HTML-Tags* entfernt wurden und nur der Text-Rest übrig geblieben ist. Zweitere beinhaltet das komplette DOM, nachdem es fertig geladen wurde. Also nicht den ursprünglichen Quelltext, sondern das endgültige Ergebnis nachdem auch alle Skripte darüber gelaufen sind und das DOM entsprechend manipuliert haben.

Interessant dürfte auch sein, den *ViewPort* der Seite entsprechend zu konfigurieren, wenn man beispielsweise eine *Responsive-Website* testen möchte:

```
page.viewportSize = {
  width: 300,
  height: 500
}
```

Allerdings liefert hier der Screenshot eine Seite, die bis zum kompletten Seitenende nach unten über den *ViewPort* hinausläuft.

Möchte man hingegen nur einen Ausschnitt der Seite haben, so kann man dies auf andere Weise angeben:

```
page.clipRect = {
  top: 50,
  left: 100,
  width: 200,
  height: 200
}
```

Hier wird ein 200 x 200 Pixel großes Quadrat, 50 Pixel von oben und 100 Pixel von links ab der oberen linken Ecke ausgeschnitten. Auf den textuellen Inhalt (*plainText* und *content*) hat dies keinen Einfluss, beziehungsweise nur in sofern, wie das DOM eventuell anders aufgebaut wird, wenn es im *responsive*-Modus angezeigt wird.

Wenn man jedoch nicht bloß Bilder oder Textausgaben der Seite möchte, ist die *evaluate*-Methode des *page*-Objektes die wichtigste Funktion. Mit dieser kann man beliebigen *JavaScript*-Code auf der Seite ausführen und somit Teile des DOMs manipulieren oder extrahieren.

Per Skript wollen wir nun alle Namen der MATHEMA-Mitarbeiter ermitteln:

```
var page = require('webpage').create(),
  url =
    "http://www.mathema.de/unternehmen/mitarbeiter";

page.open(url, function(status) {

  var mitarbeiter = page.evaluate(function() {
    var result = [];
    console.log(
      "Sammle H1-Elemente mit Mitarbeiternamen"
    );
    var elements =
      document.querySelectorAll(
        "#employees div.employee h1"
      );
    for (var i = 0; i < elements.length; i++) {
      result.push(elements[i].textContent.trim());
    }
    return result;
  });

  console.log(mitarbeiter);

  phantom.exit();
});
```

Dazu übergeben wir der *evaluate*-Methode eine Funktion, welche die Anweisungen enthält, die auf der Seite ausgeführt werden. Der letzte Teil ist besonders wichtig! Wir kommen gleich nochmal darauf zu sprechen.

In unserem Fall holen wir uns also das *document*-Objekt mit dem wir eine *JavaScript*-Referenz auf die Seite haben. Über die Methode *querySelectAll* holen wir uns dann mittels DOM-Kaskade genau die HTML-Elemente in denen die Namen stehen. Hier sind das *h1*-Elemente. Da wir jedoch nicht die HTML-Elemente selbst, sondern nur deren textuellen Inhalt benötigen, iterieren wir noch über dieses Array, holen uns den *textContent* und trimmen jeglichen *Whitespaces* drumherum weg. Die Ergebnisse pushen wir in ein Rückgabe-Array, welches als Ergebnis unseres Skripts auf der Konsole ausgegeben wird.

Wer das Skript selbst ausprobiert hat, der wird sich vielleicht gewundert haben, wo unsere *Log*-Ausgabe innerhalb der *evaluate*-Methode geblieben ist. Hier kommt der betonte Teil von oben zum Tragen. Diese *Log*-Ausgabe wird nicht auf der Konsole von *PhantomJS*, sondern auf der Konsole des intern laufenden Browsers, also genau da wo unsere *evaluate*-Anweisungen ausgeführt werden, ausgegeben. Blöd, da diese Konsole ja nie sichtbar wird.

Der Trick ist diese Konsolenausgaben zu fangen und an die *PhantomJS*-Konsole umzuleiten. Dies gilt dann auch für die eventuellen Ausgaben der Skripte auf der Seite selbst. Dazu bietet das *page*-Objekt die Möglichkeit einige Callback-Methoden [1] zu registrieren, unter anderem für *console*-Nachrichten:

```
page.onConsoleMessage = function(msg, lineNum, sourceId) {
  console.log('CONSOLE: ' + msg);
};
```

Dasselbe Problem haben wir dann auch mit Variablen. Innerhalb der *evaluate*-Funktion sind keine zuvor registrierten Variablen des *PhantomJS*-Skriptes mehr verfügbar. Dafür hat man jedoch Zugriff auf globale Objekte der Seite, also alle Objekte die durch Skripte erzeugt werden. Wie zum Beispiel das *jQuery*-Objekt, wenn dieses in der Seite eingebunden wurde.

Sollte man jedoch trotzdem Bedarf haben, programmatisch erzeugte Werte des *Phantom*-Skriptes innerhalb des Browsers zu benötigen, so kann man diese einfach als Übergabeparameter hereinreichen. Angenommen unser

Selektor, den wir in der Seite benutzen, wurde zuvor erzeugt, dann sähe das Skript so aus:

```
page.open(url, function(status) {
  var selector = "#employees div.employee h1";

  var elements = page.evaluate(function(sel) {
    return document.querySelectorAll(sel);
  }, selector);

  console.log(JSON.stringify(elements));

  phantom.exit();
});
```

Man beachte den zweiten Übergabeparameter an die *evaluate*-Methode, nachdem die ganze Funktion fertig definiert wurde! In diesem Skript sieht man auch, dass man die Ergebnisse ohne Probleme in *JSON-Notation* aufbereitet zurückliefern kann. So können in der umgebenden Plattform, welche das *PhantomJS*-Skript einbindet, wiederum einfacher Auswertungen gefahren werden, ohne dass man das Ergebnis erst mühsam parsen muss.

Sub-Seiten

Zum Abschluss wollen wir noch testen wie man weitere Seiten öffnen kann und Dateien speichert.

Zu einigen Mitarbeitern gibt es bereits eigene Infoseiten, welche wir öffnen und einen Screenshot davon erstellen wollen. Anschließend wollen wir die Liste aller Mitarbeiter einfach in einer Datei speichern. Das fertige Skript würde wie folgt aussehen.

```
var webpage = require('webpage'),
  fs = require('fs'),
  page = webpage.create(),
  url = "http://www.mathema.de/unternehmen/mitarbeiter";

page.open(url, function(status) {
  var subPage, i, href,
    selector = "#employees div.employee h1";

  //Mitarbeiter-Links ermitteln
  var links = page.evaluate(function(sel) {
    var result = [],
      elements = document.querySelectorAll(sel+" a");
    for (var i = 0; i < elements.length; i++) {
      result.push(elements[i].href);
    }
    return result;
  }, selector);

  //Links öffnen und speichern
  for (i = 0; i < links.length; i++) {
    href = links[i];
    subPage = webpage.create();
    subPage.open(href, function(status) {
      window.setTimeout(function() {
        subPage.render(
          href.substring(href.lastIndexOf("/")+1)+".png"
        );
      }, 1000);
    });
  }

  //Mitarbeiterliste speichern
  var mitarbeiterListe = page.evaluate(function(sel) {
    var result = [],
      elements = document.querySelectorAll(sel);
    for (var i = 0; i < elements.length; i++) {
      result.push(elements[i].textContent.trim());
    }
    return result;
  }, selector);
  fs.write("liste.txt", JSON.stringify(mitarbeiterListe), "w");

  phantom.exit();
});
```

Im ersten Teil ermitteln wir wie gehabt eine Selektion auf der Seite; in diesem Fall mögliche Links zu einem Mitarbeiter. Von diesen Links merken wir uns die URLs.

Im zweiten Schritt iterieren wir über das fertige Array aus URLs und öffnen diese entsprechend in neuen Tabs. Von denen wir Screenshots erstellen und unter dem Namen des letzten URL-Stücks abspeichern. Das Öffnen von neuen Tabs gestaltet sich leider in *PhantomJS* noch sehr hakelig, weswegen wir den *Render*-Schritt per *Timeout* verzögern. Der Grund hierfür ist, dass *PhantomJS* im Falle von Unterseiten leider nur noch sehr unzuverlässig ermitteln kann, ob die Seite denn wirklich schon fertig geladen wurde. Leider hakte im Test auch das *Rendern* der Unterseiten.

Im letzten Schritt ermitteln wir, wie schon zuvor, die Liste der Mitarbeiter. Um diese in einer Datei zu speichern, benötigen wir zusätzlich das *fs*-Modul [2] und dessen *write*-Methode. Hier gibt man zunächst einen Dateinamen, dann den Inhalt und zum Schluss einen Modus an. *w* bedeutet, dass die Datei immer vollständig geschrieben wird und eventuell vorhandene Dateien ersetzt. *a* hingegen fügt nur hinzu.

Ausblick

Eine weitere Idee wäre die Mitarbeiter-Bilder zu grabben und abzuspeichern. Doch hier stößt man dann langsam

an die Grenzen von *PhantomJS*. Datei-Downloads sind anscheinend momentan nicht vorgesehen. An die Bilder würde man mit Tricks allerdings noch herankommen, indem man durch geschickte Positionierung des *Clipping*-Rechtecks und der *Render*-Methode quasi Screenshots der Fotos erstellt. Dies wäre dann für einen Einblick in *PhantomJS* jedoch zu weit gegangen. Auch haben wir die Möglichkeiten zur Netzwerkanalyse ausgelassen, oder das *SVG-Rendering*, welches *PhantomJS* dank der *WebKit-Engine* sehr gut beherrscht.

Eine 2.0-Version von *PhantomJS* ist auch in Arbeit, in der eine neuere Version der zugrundeliegenden *Qt*- und *QtWebkit*-Bibliothek verwendet werden soll.

Wer längere Navigationspfade testen möchte, kann sich außerdem auch *CasperJS* anschauen, welches auf *PhantomJS* aufsetzt und *CoffeeScript* unterstützt.

Referenzen

[1] PhantomJS *API – Web Page Module*, http://phantomjs.org/api/webpage

[2] PhantomJS *API – File System Module*, http://phantomjs.org/api/fs

Weiterführende Literatur

- PhantomJS
  http://phantomjs.org
- CasperJS
  http://casperjs.org

Kurzbiografie

Frank Goraus (frank.goraus@mathema.de) ist Senior Developer bei der MATHEMA Software GmbH in Erlangen. Seit 2006 beschäftigt er sich bereits mit der Entwicklung von JEE-Anwendungen, u. a. in Verbindung mit einem Portal-Server. Seine Liebe zum Detail verwirklicht er mit seinen Web-Design-Kenntnissen. In seiner Freizeit beschäftigt er sich außerdem mit Android-Entwicklung, verschiedensten Web-Frameworks und einem eigenen Projekt für eine Sammlungsverwaltung.

Auf Safari im Programmier-dschungel

JavaScript auf der JVM mit Nashorn

von Marc Spanagel

JavaScript ist auf dem Vormarsch. Seit Java 8 ist es nun auch fester Bestandteil des JDK und hört auf den deutschen Namen *Nashorn*. Dynamisches JavaScript auf der JVM – wie funktioniert das denn?

Mit Einführung des JDK 8 wurde auch eine *JavaScript-Engine* namens *Nashorn* umgesetzt, welche die teils inperformante Referenzimplementierung *Rhino* von Mozilla ablöst.

Die Performanz soll letzteren um den Faktor 2 – 10 übertreffen, laut Lead-Entwickler Lagergren nach oben noch nicht ausgereizt. Die Entwickler haben es auch geschafft, volle 100% ECMA-Konformität zu erreichen.

Im Gegensatz zu *Rhino* stand den *Nashorn*-Entwicklern jedoch der JSR 292 (*Supporting Dynamically Typed Languages on the JavaTM Platform* [1]) aus dem JDK 7 zur Verfügung, welcher mittels *invokedynamic* die Anbindung dynamischer Sprachen an die JVM ermöglicht bzw. vereinfacht.

Was ist nun dieses *invokedynamic*, von dem der Leser sicher schon einmal gehört hat? Hierzu ein kleiner Exkurs.

Java ist bekanntermaßen eine statisch typisierte Sprache – jede Variable muss einen bestimmten Typ besitzen, jede Methode einen Typ zurückgeben (ja, auch *void* ist ein Typ). Das *Casten* von typisierten Variablen wird vom *Compiler* überwacht.

Auch der Bytecode hält die Typisierung aufrecht und diese wird wiederum von der JVM überwacht.

Im Gegensatz dazu stehen dynamische Sprachen, die das Konzept des *Ducktypings* verfolgen:

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. [2]*

Dies meint, dass man auf jedem (dynamischen) Objekt beispielsweise die Methode *quak()* aufrufen kann, sofern man sich sicher ist (oder zumindest glaubt), dass es sich bei dem Objekt um eine Ente handelt. Für den Compiler bedeutet das, dass er die Entscheidung, welches die passendste, aufzurufende Methode ist, in die Laufzeit verschieben muss. Zur *Compile*-Zeit weiß er ja nicht, ob es sich bei dem Objekt um eine Ente, ein Rüsselschwein oder Hänschenklein handelt.

Und hier kommt *invokedynamic* ins Spiel. Der kundige Leser kennt bestimmt die *JIT-Bytecode*-Instruktionen wie *invokestatic* oder *invokevirtual*, welche die JVM instruieren Klassen- oder Objektmethoden aufzurufen, die alle mit der JVM fest verdrahtet sind. Im Gegensatz dazu gibt eine einzelne *Invokedynamic*-Instruktion, auch *Dynamic Call Site* genannt, der JVM den Hinweis im Falle einer Operation nach einer sogenannten *Bootstrap*-Methode zu suchen, welche wiederum auf eine oder mehrere Implementierungen der Operation mit Hilfe von *MethodHandles* verweist. Referenzen zu diesen *Bootstrap*-Methoden werden in dem sogenannten *Constant Pool* gehalten und mit den *invokedynamic*-Einträgen verbunden. Die jeweiligen konkreten Implementierungen muss der Konstrukteur der dynamischen Sprache als Bibliothek mitliefern.

Im Prinzip heißt das, der Bytecode wird angereichert durch externe Informationen, die zur Laufzeit entscheiden, welche Methode auszuführen ist. Genaueres kann unter [3] nachgelesen werden.

Doch kommen wir zurück zu unserer kleinen Tierschau. Manch ein Leser mag schon bei der Einleitung gedacht haben: Nicht schon wieder ein JavaScript-Framework. Doch im Gegensatz zu manch anderen Frameworks ist *Nashorn* nicht dazu gedacht, das DOM auf Client-Seite innerhalb einer HTML-Seite im Browser zu manipulieren – tatsächlich wird dies überhaupt nicht unterstützt. Ziele bei der Entwicklung von *Nashorn* waren, neben einem *Proof of Concept*, die JVM für dynamische Sprachen allgemein zu öffnen und unter anderem Java-Code aus JavaScript und umgekehrt aufzurufen.

Grundsätzlich kann *Nashorn* auf zwei Arten ausgeführt werden:

- auf Skriptebene mittels *jjs* oder *jrunscript*
- innerhalb einer Java-Anwendung mittels der *javax.script.ScriptEngine*

Beides werden wir im Folgenden betrachten.

1. Skripting

Auf Skriptebene wird JavaScript-Code ausgeführt, welcher auch Java-Objekte erzeugen und verarbeiten kann. Im *bin*-Ordner des JDK 8 findet man das Tool *jjs*, welches die *Nashorn*-Engine aufruft. Alternativ existiert dort auch das aus dem JDK 7 bekannte *jrunscript*, dies ist jedoch experimentell – mit ihm lassen sich über den Startparameter –l auch andere Skriptsprachen ausführen (welche als *JAR*-Dateien vorliegen und natürlich mit JAR-223 konform sein müssen – standardmäßig ist JavaScript voreingestellt). Ebenso lassen sich Skripte im *Batchmode* (über den Parameter *–f*) ausführen. Lassen wir ein paar Beispiele folgen:

```
jjs> print("Bitte kein 'Hello World' mehr...")
Bitte kein 'Hello World' mehr...

jjs> function add(a,b) a+b;
function add(a,b) a+b;
jjs> add(1,2)
3
jjs> add(1,"2")
12
```

Java-Funktionen und -Typen können sehr einfach aufgerufen und erzeugt werden:

```
jjs> var foo = "12345".substring(2)
jjs> print(foo)
345

jjs> var list = new java.util.ArrayList();
jjs> list.add(123)
true
jjs> list.add(456)
true
jjs> print(list)
[123, 456]
```

Nicht möglich und sinnvoll ist natürlich die Verwendung von *Generics* – dies macht bei dynamischen Sprachen ja auch keinen Sinn, da die Informationen zur Laufzeit sowieso wieder verschwinden und es dem Compiler ja, wie oben erwähnt, schnurz ist.

Scripting-Mode

Startet man *jjs* mit dem Parameter *-scripting*, befindet man sich im Skripting-Modus, der es ermöglicht weitere Spracherweiterungen wie *heredocs* und *shell invocations* zu nutzen.

Heredocs

Heredocs sind mehrzeilige Zeichenfolgen, deren Formatierung erhalten bleibt und innerhalb dessen JavaScript-Variablen und -Funktionen ausgewertet werden.

Der Beginn einer *Heredoc* wird durch *<<EOF* gekennzeichnet, das Ende durch *EOF*. Diese Datei *heredoc.js*

```
print(<<EOF
foooo ${new Date()}
  bar
EOF);
```

wird durch den Aufruf von *jjs* mit den Argumenten

```
$ jjs -scripting heredoc.js
```

ausgegeben als (man beachte den Zeilenumbruch und die Einrückung):

```
foooo Fri Nov 28 2014 17:36:15 GMT+0100 (CET)
  bar
```

Den Parameter *-scripting* kann man sich sparen, wenn man, wie in *Unix*-Skripten gebräuchlich, einen *Shebang* angibt – für *Nashorn* also *#!/usr/bin/jjs*.

Shell-Aufrufe

Mit *shell invocations* können externe Programme aufgerufen und deren Ergebnisse weiterverarbeitet werden, was eine ziemlich coole Angelegenheit ist. Wer hat nicht schon einmal auf *Linux*-Kisten kompliziertere Skripte benötigt, aber diverse *Flags*, Argumente und Kommandos wieder vergessen, wegen des seltenen Gebrauchs. Diese wieder in das Gedächtnis zu rufen und zu suchen, hat zwar auch seinen eigenen Reiz, kostet aber viel Zeit, während das Wissen in Java(Script) meist omnipräsent ist.

Das Beispiel dieses Scripts hier ruft alle Dateien im aktuellen Verzeichnis auf (was in Linux auch *awk* erledigen könnte – wenn es dem Autor nur gerade einfiele):

1. `var dir= \`cmd.exe /k dir\`.split("\n");// -->Windows`

oder

2. ```
   var dir= `ls`.split("\n");// -->Linux
   for each (var entry in dir) {
     var arr = entry.split(" ");
     print(arr[arr.length-1]);
   }
   ```

Eine Anmerkung hierzu: Die *back-tick*-Striche, die das auszuführende Kommando umschließen, sind die auf der deutschen Tastatur links neben der *Backspace*-Taste bei gedrücktem *Shift*.

Daneben existieren noch globale Objekte im Skripting-Modus, mit denen man beispielsweise Zugriff auf den Standard-Output ($OUT) hat oder ebenfalls externe Programme ausführen kann ($EXEC). Der interessierte Leser sei auf [4] verwiesen.
   ```

ScriptEngine

Skripte können in Java-Anwendungen aufgerufen werden, indem man sich die passende Script-Engine erzeugt, entweder über die *ScriptEngineFactory*, genauer gesagt die *NashornScriptEngineFactory*, oder genauso einfach über den *ScriptEngineManager*, der sich einfach alle verfügbaren *ScriptEngineFactories* schnappt und dann die Engine über den Namen herausziehen lässt.

Skripte können schließlich über die *eval*-Methode der Engine ausgeführt werden, entweder als *String* oder über einen *Reader*:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class Zoo {
  public static void main(String... args)
  throws Exception {
    ScriptEngineManager manager =
      new ScriptEngineManager();
    ScriptEngine engine =
      manager.getEngineByName("nashorn");
    String js = "print('foo')";
    engine.eval(js);
```

Will man nur einzelne Funktionen eines Skripts aufrufen, muss man sich des *Invocable*-Interfaces bedienen – die *Nashorn-Engine* implementiert dies, jedoch wird es anderen Engines nicht vorgeschrieben, so dass man sich entweder sicher sein muss, dass ein *Cast* auch gelingt oder gleich die explizite Engine, hier also die *Nashorn-ScriptEngine*, deklariert.

```
ScriptEngineManager manager =
  new ScriptEngineManager();
ScriptEngine engine =
  manager.getEngineByName("nashorn");
String js = "var add = function(a, b) a+b";
engine.eval(js);
Object result =
  ((Invocable) engine).invokeFunction("add", 2, 3);
  engine.eval("print(" + result.toString() + ")");
```

Bindings und ScriptContext

Um Objekte für Skript zur Verfügung zu stellen, existiert unter anderem das Konzept der *Bindings*.

javax.script.Bindings sind eigentlich nur eine Map mit einem *String* als Schlüssel und existieren in zwei Scopes:

- GLOBAL_SCOPE
- ENGINE_SCOPE

Beim Erzeugen der Engine wird standardmäßig ein *ScriptContext* mit dem Scope ENGINE_SCOPE erzeugt, der aber tatsächlich dem globalen ECMA-*Scope* entspricht.

Im globalen ECMA-Scope sind, wie der Name schon sagt, globale oder Standardobjekte enthalten, wie beispielsweise *Object, Function* etc. [5]. Es können aber auch neue, eigene Objekte hinzugefügt werden.

Im Browser wäre ein globales Objekt etwa *window*. Deswegen ist es auch möglich, in JavaScript *window.open(...)* auszuführen, denn der *global scope* kennt hier das *window*-Objekt.

Übrigens sind die globalen Objekte (die im *global scope*) nicht zu verwechseln mit dem *global object* (*this* im *global scope*) – denn der *global scope* besteht aus den Properties dieses globalen Objekts.

Warum einfach, wenn es auch kompliziert geht.

Der GLOBAL_SCOPE ist wohl eher für User-spezifische Objekte gedacht – denn wenn Objekte nicht im ENGINE_SCOPE gefunden werden, werden sie anschließend im GLOBAL_SCOPE gesucht – versehentliche Überschreibungen von globalen Objekten können somit nicht so einfach passieren.

Das folgende Skript gibt einige globale Objekte aus.

```
ScriptContext defaultCtx = engine.getContext();
Bindings bindings =
  defaultCtx.getBindings(ENGINE_SCOPE);
System.out.println(bindings.get("Object"));
// Ausgabe: function Object() { [native code] }

System.out.println(bindings.get("Array"));
// Ausgabe: function Array() { [native code] }

System.out.println(bindings.get("NaN"));
// Ausgabe: NaN

System.out.println(bindings.get("eval"));
//Ausgabe: function eval() { [native code] }
```

Zur Demonstration der verschiedenen Scopes dient das folgende Programm:

```
ScriptEngineManager manager =
  new ScriptEngineManager();
ScriptEngine engine =
  manager.getEngineByName("nashorn");
String js = "print(arbeitenOderNicht)";
ScriptContext defaultCtx = engine.getContext();
defaultCtx.getBindings(GLOBAL_SCOPE).put(
  "arbeitenOderNicht", "Faul sein ist wunderschön!"
);
engine.eval(js);
/* -> Kein explizites Binding angegeben,
 * Nashorn sucht zuerst im ENGINE_SCOPE
 * und dann im GLOBAL_SCOPE.
 * Ausgabe: Faul sein ist wunderschön!
 */
```

```
defaultCtx.getBindings(ENGINE_SCOPE).put(
   "arbeitenOderNicht", "Arbeit macht das Leben süß!"
);
engine.eval(js);
/* -> Kein explizites Binding angegeben,
 * Nashorn sucht zuerst im ENGINE_SCOPE
 * und wird bereits hier fündig
 * Ausgabe: Arbeit macht das Leben süß!
 */

ScriptContext ctx = new SimpleScriptContext();
Bindings globalBinding = new SimpleBindings();
globalBinding.put(
   "arbeitenOderNicht",
   "Arbeit macht das Leben süß, Faulheit stärkt die Glieder!"
);
ctx.setBindings(globalBinding, ENGINE_SCOPE);
engine.eval(js, ctx);
/*
 * Explizites Binding angegeben,
 * Ausgabe: Arbeit macht das Leben süß, Faulheit
 * stärkt die Glieder!
 */
```

Fazit

Die Integration von JavaScript in Java bietet interessante Möglichkeiten. Mittels *Shell*-Aufrufen Programme des Betriebssystems zu starten und dann vorallem deren Ergebnisse mittels JavaScript weiterverarbeiten zu können, ist nur ein guter Ansatz.

Auch dynamischen Code innerhalb von Java-Anwendungen auszuführen, bietet interessante Möglichkeiten. Beispielsweise könnte kundenspezifischer Code dynamisch geladen werden, oder man könnte Schnittstellen öffnen, um bestehenden Code dynamisch zu erweitern.

Es wird sicher spannend mitzuerleben, welche Applikationen und Einsatzbereiche, analog *Node.js*, mittels *Nashorn* noch entwickelt und entdeckt werden.

Referenzen

[1] Java Community Process *JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform,* https://www.jcp.org/en/jsr/detail?id=292

[2] Wikipedia *Duck typing* http://en.wikipedia.org/wiki/Duck_typing

[3] ORACLE *Java Virtual Machine Support for Non-Java Languages* https://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html

[4] ORACLE *4 Nashorn and Shell Scripting* http://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/shell.html

[5] MDN mozilla developer network *Standard built-in objects* http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Kurzbiografie

Marc Spanagel (marc.spanagel@mathema.de) ist als Senior Consultant für die MATHEMA Software GmbH tätig. Sein Schwerpunkt liegt auf der Entwicklung mit JEE und C#. Daneben fasziniert ihn die 3D-Programmierung.

Des Programmierers kleine Vergnügen

Ausbruchssicher

von Michael Wiedeking

Versucht man auf ein *Array*-Element zuzugreifen, das nicht existiert, so wird dies in nicht ganz so toleranten Sprachen zurecht mit etwas wie einer *IndexOutOfBounds*-Ausnahme geahndet. Gelegentlich würde es aber schon genügen, wenn auch noch beim Verlassen des gültigen Bereichs ein Element geliefert würde.

Dass *Array*-Grenzen geprüft werden ist ja eigentlich selbstverständlich. Damit also kein Schindluder getrieben wird, wird ein solcher Grenzverstoß beispielsweise mit einer Ausnahme quittiert. In manchen Fällen kann es aber nützlich sein, dass bei einem Verlassen des gültigen Bereichs trotzdem ein Wert aus dem Array geliefert wird.

Allgemein könnte bei einer zusammenhängenden Sequenz von ganzen Zahlen, wenn MIN davon die kleinste und MAX die größtmögliche zulässige Zahl ist, beim Unterschreiten des Bereichs immer MIN und beim Überschreiten immer MAX geliefert werden. Naiv lässt sich dieser Ansatz exemplarisch für eine Indexmenge eines Arrays wie folgt definieren:

```
int indexOf(int i) {
  if (i < MIN) {
    return MIN;
  } else if (i > MAX) {
    return MAX;
  } else {
    return i;
  }
}
```

Es geht aber auch ein bisschen mathematischer, indem man Minimum- und Maximumfunktion bemüht:

```
int indexOf(int i) {
  return min(max(i, MIN), MAX);
}
```

Wenn der Prozessor bedingungsfreie Varianten anbietet, ist das auch die schnellstmögliche Lösung. Apropos bedingungsfrei: Unter bestimmten Bedingungen mag es sich sogar lohnen, die aus dem kleinen Vergnügen [1] hergeleitete bedingungsfreie Variante anzuwenden.[1]

Wenn die Größe der Indexmenge eine Zweierpotenz ist und dementsprechend von 0 bis $2^k - 1$ geht, ergibt sich – wie schon so oft – besonderes Optimierungspotenzial. Das liegt daran, dass hier nur die niederwertigen k Bits gültig sind. Ist auch nur eines der höheren Bits gesetzt, dann ist klar, dass der Index außerhalb liegt. Betrachtet man also

$$h = i \gg k$$

verbleiben nur noch die höherwertigen Bits h. Ist $h = 0$, dann ist der Index korrekt und es muss nichts gemacht werden. Im anderen Fall bleibt dankbarerweise sogar noch das Vorzeichen erhalten. Denn so lässt sich leicht herausfinden, ob der niedrigste oder höchste Index berechnet und geliefert werden muss. In diesem Fall benötigt man also für $h < 0$ den niedrigsten Index, also 0, und für $h \geq 0$ den höchsten Index, von dem wir ja wissen dass er die Form $2^k - 1$ hat.

Ist h positiv, so benötigt man $2^k - 1$, also eine Zahl, bei der alle niederwertigen k Bits gesetzt sind. Die bekommen wir üblicherweise dadurch, dass wir zunächst das n-Bit-Wort nehmen, bei dem alle n Bits gesetzt sind, und das ist -1. Verschiebt man dieses nun ohne Berücksichtigung des Vorzeichens um $n - k$ Stellen, so erhält man die gewünschte Zahl. Statt -1 kann man aber auch h nehmen. Dort sind nämlich genau k Nullen oder Ein-

[1] In der ich leider die Definitionen für *min* und *max* vertauscht habe. Richtig müsste es dort also heißen: $\min(x, y) = x - \mathrm{doz}(x, y)$ und $\max(x, y) = y + \mathrm{doz}(x, y)$.

sen nachgeschoben worden; in diesem Fall k Nullen. Invertiert man h mit $\sim h$, bekommt man die gewünschten k Einsen, die man nur noch an die richtige Stelle schieben muss. Dabei ist darauf zu achten, dass nur Nullen nachgeschoben werden:

```
i = ~h >>> (n – k);
```

Ist aber h negativ, benötigt man die 0. Das kann man erreichen, indem man h invertiert und alle unnötigen Bits rausschiebt, so dass nur noch Nullen verbleiben. Im positiven Fall hat man sich ja h bedient. Das kann man hier analog machen, nur dass im negativen Fall natürlich k Einsen an den hochwertigen Stellen zu finden sind. Invertiert man diese, so verbleiben k Nullen. Schiebt man nun die niederwertigen $n - k$ Stellen heraus, so erhält man die gewünschte 0.

Damit kann man den Aufwand für MIN = 0 und MAX = $2^k - 1$ auf jeden Fall schon einmal auf einen einzigen Test reduzieren:

```
static int indexOf(int i) {
  int h = i >> k;
  if (h ≠ 0) {
    i = ~h >>> (32 – k);
  }
  return i;
}
```

Wie schon in mehreren Vergnügen zu sehen war, lässt sich $h \neq 0$ bedingungsfrei als $b = h \mid (-h)$ bestimmen, wenn das gesetzte Vorzeichen-Bit in b als *true* interpretiert wird. Verschiebt man dieses nun vorzeichenbehaftet bis in die niederwertigste Stelle, so erhält man bei $h \neq 0$ den Wert –1 und andernfalls den Wert 0.

```
int b = (h | (–h)) >> 31;
```

Nun hat man eine Maske, die auf alle Bits angewandt werden kann. Für $h = 0$ und damit $b = 0$ möchte man i unverändert; im anderen Fall ist $h \neq 0$ mit $b = -1$ und der Index muss wie oben gesehen beschränkt werden. Das lässt sich einfach bewerkstelligen, indem man b und das invertierte $\sim b$ mit dem jeweils gewünschten Wert über ein Bit-weises *Und* verknüpft und deren Ergebnisse mit einem *Oder*. Damit erhält man

```
static int indexOf(int i) {
  int h = i >> k;
  int b = (h | (–h)) >> 31;
  int j = (~h >>> (32 – k))
  return (i & (~b)) | (j & b);
}
```

Dabei lässt sich der *return*-Ausdruck noch um eine Instruktion vereinfachen, indem man diesen durch

```
return i ^ ((i ^ j) & b)
```

ersetzt. Ist $b = 0$, so verbleibt einfach nur i, weil der geklammerte Ausdruck 0 wird; andernfalls bleibt die Klammer erhalten und ergibt wegen $i \wedge i = 0$ wunschgemäß j.

Aber Achtung: Hier muss sorgfältig getestet werden, ob diese zehn Instruktionen überhaupt eine Chance gegen obige *min-max*-Variante haben. Die braucht nämlich auch im allgemeinen Fall, wenn sie gemäß [1] „zusammengefrickelt" wird, nur maximal 8 Instruktionen, während diese Version schon 10 braucht. Das lässt sich zwar bei konstantem k noch um eine Instruktion vermindern, aber das würde ja auch bei der *min-max*-Version zur Reduktion um mindestens eine Instruktion führen.

Letztlich hängt es von der Struktur des Prozessors ab, ob nicht sogar die Variante mit dem einen *if* konkurrenzfähig ist. Bei Hochsprachen tritt dann ja noch zusätzlich das Problem auf, dass bei all zu geschickten Frickeleien der *Compiler* oft nicht erkennen kann, was man eigentlich wollte. So hat bei Java bedauerlicherweise gelegentlich schon die Einführung oder das Entfernen von Hilfsvariablen einen (nicht vorhersehbaren) Effekt auf die spätere Optimierung durch den *Hot-Spot-Compiler*.

Demnach hat die *min-max*-Variante die wohl größten Chancen, in verschiedenen Umgebungen ein sehr gutes (wenn nicht sogar das beste) Ergebnis zu liefern – was insbesondere im Bezug auf die Lesbarkeit ein nicht zu vernachlässigender Punkt ist.

Referenzen

[1] Wiedeking, Michael *Des Programmierers kleine Vergnügen – Differenz oder Null – noch einmal*, Jahrgang 7, Nr. 4, S. 15, Bookware, April 2014 http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2014-04.pdf

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

Kaffeesatz

Rise of the Machines

von Michael Wiedeking

Das fahrerlose Auto kommt – soviel ist sicher. Und dieser Fortschritt macht auch nicht vor Rennwagen halt. So fuhr neulich ein 560 PS starker Audi RS 7 ohne Fahrer auf der Rennstrecke der Motorsport Arena Oschersleben mit einem baugleichen, normal bemannten Exemplar um die Wette. Um das vorweg zu nehmen: Der Mensch hat gewonnen. Für die 3,7 km lange Strecke benötigte der Sieger 1:58,4 Minuten, während die unbeseelte Maschine 2:21,6 Minuten brauchte.

Der eine oder andere wird gleich einwerfen, dass eine Rennstrecke, auf der die beiden Kontrahenten noch nicht einmal gleichzeitig fahren, wahrlich keine Herausforderung ist: Kein Gegenverkehr, keine Konkurrenz, keine Fußgänger, keine Hindernisse und keine Jugendlichen, die spaßeshalber einen großen Gummiball auf die Fahrbahn werfen, um den Verkehr zum Erliegen zu bringen. Aber sehr lange wird man sich nicht über die Unfähigkeit der Maschinen amüsieren können.

Viele erinnern sich wahrscheinlich noch an den Schachcomputer Deep Blue, der am 10. Februar 1996 gegen den damals amtierenden Schachweltmeister Garri Kimowitsch Kasparow spielte – und gewann. Das erste Mal in der Geschichte der Schachcomputer gewann ein solcher gegen einen Großmeister unter Turnierbedingungen. Es war eine Sensation und doch ungemein befriedigend, dass letztendlich der Großmeister mit 4:2 die Oberhand behielt.

Nur 14 Monate später siegte Deep Blue mit 3,5:2,5 über den Meister und schubste damit den schachspielenden Menschen vom Thron des „königlichen Spiels“. Was der Mensch spätestens seit der Version 1.0 der Schachautomaten, dem 1769 von dem österreichisch-ungarischen Hofbeamten und Mechaniker Wolfgang von Kempelen konstruierten „Schachtürken“, erträumte, wurde damit Wirklichkeit. Zugegebenermaßen wurde bei der Version 1.0 noch „getürkt“, versteckte sich doch ein Schachspieler in dem Automaten. Allerdings ging es auch beim endlichen Sieger nicht viel besser zu, durfte doch der Code (vereinbarungsgemäß) zwischen den Spielen angepasst werden.

Nach nur zwölf Jahren Entwicklungszeit konnte die Mannschaft um Deep Blue ihr Ziel – mit Hilfe eines Rechners einen Großmeister zu besiegen – in die Tat umsetzten. Schachspielen ist sicherlich nicht trivial, aber es ist eindeutig geregelt, damit vorhersehbar, offensichtlich modellier- und berechenbar; und vor allem bewegt sich das Schachbrett (das der Rechner ja gar nicht benötigt) nicht von der Stelle. Autofahren ist da ungleich komplizierter. Aber diese Komplexität zu beherrschen ist nur eine Frage der Zeit. Und seit damals sind fast 20 Jahre vergangen.

Übrigens ist das „königliche“ Schach schon lange nicht mehr das Maß aller Dinge. Die neue große Herausforderung ist das Brettspiel *Go*. Statt der lächerlichen 20 möglichen Anfangszüge beim Schach, bietet ein normales 19 × 19 Feld schon 361 Möglichkeiten. Nach nur zwei Zügen beim Schach gibt es zwar schon 72 084 verschiedene Stellungen, und bei 40 Zügen kommt man schätzungsweise auf 10^{115} bis 10^{120} verschiedene Spielverläufe. Aber das *Go*-Brett ist fast sechsmal so groß.

Da *Go* deswegen nicht mit der brachialen Gewalt der meisten Schachcomputer beizukommen ist, müssen schon clevere Algorithmen her. Aktuell macht man sich in der Universität Edinburgh Gedanken darüber, wie man sich dem Spiel mit neuronalen Netzen nähern kann. Angeblich kann sich das Ergebnis auch schon sehen lassen, denn es schlägt *GNU Go* in fast 90 % der Fälle. Für diejenigen, denen folgendes etwas sagt: *GNU Go* hat einen geschätzten Rang von 6 – 8 Kyū. Für die, denen das nichts sagt: Anfänger steigen bei einem Rang von 20 – 30 Kyū ein, der 1. Kyū ist also der höchste.

Ach, aber irgendwie ist es doch egal: Zuallererst sind die Menschen unter sich, dann findet sich ein ehrgeiziger Entwickler, der was auch immer mit dem Computer machen will. Zunächst hat der Computer gegen den Menschen keine Chance, dann, in der Übergangsphase, kann der Mensch noch mitmischen, bis er schließlich irgendwann vom Computer so weit abgehängt wird, dass er lieber wieder unter seinesgleichen bleibt. Dann spielen die Computer gegeneinander, aber nicht etwa weil sie das wollen sondern um das Ego der Entwickler zu befriedigen.

Naja, solange wir wissen, wozu das alles gut ist…

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.

Bookware, Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0, Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

Java UserGroup Bremen
http://www.jugbremen.de
Kontakt: Rabea Gransberger (rgransberger@gmx.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: jug-da-orga@googlegroups.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Falk Hartmann
(falk.hartmann@jugsaxony.org)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o Stefan Koospal
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
http://jug-karlsruhe.de
(jugkarlsruhe@gmail.com)

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Natalia Wilhelm
(info@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)
Rainer Anglett (ra@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)

.NET User Groups

Deutschland

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(info@dodnedder.de)

.NET UserGroup Frankfurt
http://www.dotnet-usergroup.de

.NET User Group Hannover
http://www.dnug-hannover.de
Kontakt: (dnug@indisoftware.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.NET User Group Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy_erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Herr Michael Niethammer

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET User Group Austria
c/o Global Knowledge Network GmbH,
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Software Craftsmanship Communities

Deutschland, Schweiz, Österreich

Softwerkskammer – Mehrere regionale Gruppen und Themengruppen unter einem Dach
http://www.softwerkskammer.org
Kontakt: Nicole Rauch (nicole.m@gmx.de)

MATHEMA

meet the experts of enterprise infrastructure

HTML5, CSS3 und JavaScript
9. – 12. März 2015, 21. – 24. September 2015
1.650,– € (zzgl. 19 % MwSt.)

Entwicklung mobiler Anwendungen mit iOS
20. – 22. April 2015, 5. – 7. Oktober 2015
1.250,– € (zzgl. 19 % MwSt.)

Fortgeschrittenes Programmieren mit Java
Ausgewählte Pakete der Java Standard Edition
4. – 6. Mai 2015, 19. – 21. Oktober 2015
1.350,– € (zzgl. 19 % MwSt.)

Weiterführende Programmierung unter C#
Umstieg auf C# 4.5/5 und Einführung in fortgeschrittene Konzepte
11. – 13. Mai 2015, 26. – 28. Oktober 2015
1.350,– € (zzgl. 19 % MwSt.)

AngularJS
18. – 19. Mai 2015, 7. – 8. Dezember 2015
950,– € (zzgl. 19% MwSt.)

Anwendungsentwicklung mit der Java Enterprise Edition
22. – 26. Juni 2015, 30. Nov – 4. Dez. 2015
2.150,– € (zzgl. 19 % MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

„Die Herausforderung, jeden Tag etwas Neues zu lernen, habe ich gesucht und bei MATHEMA gefunden."

Tim Bourguignon, Senior Consultant

Wir sind ein Consulting-Unternehmen mit Schwerpunkt in der Entwicklung unternehmenskritischer, verteilter Systeme und Umsetzung von Service-orientierten Architekturen und Applikationen von Frontend bis Backend. Darüber hinaus ist uns der Wissenstransfer ein großes Anliegen: Wir verfügen über einen eigenen Trainingsbereich und unsere Consultants sind regelmäßig als Autoren in der Fachpresse sowie als Speaker auf zahlreichen Fachkonferenzen präsent.

join the experts of enterprise infrastructure

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de | Ansprechpartnerin:
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de | Britta Kolb

Das Allerletzte

```
myPartitionToDispose = myPartition;

// I'm not going to try to maintain this
if (state != null)
    state.CurrentIteration = 0;

while (myPartition.MoveNext())
```

Dies ist kein Scherz!
Dieser Code-Abschnitt wurde tatsächlich
in der freien Wildbahn (in diesem Fall im Code
des .NET-Frameworks) angetroffen.

Ist Ihnen auch schon einmal ein Exemplar dieser
Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im Januar.

Herbstcampus

Wissenstransfer par excellence

31. August – 3. September 2015
in Nürnberg