Dart Mobile Development

Sriram Balachandran (ID: 805167463)
University of California, Los Angeles

Abstract

In this day and age, smartphones have grown almost as ubiquitous as pencils, and they serve a crucial role in everyday life. The mobile applications that they house and empower solve complex daily problems and the convenient touch of a button. As applications grow more complex and become more frequently used, developers seek to minimize latency in every way possible. One such method in which one can manipulate latency is through choice of architecture in mobile application - should complex calculation be performed in-device or delegated to external services with more computational power?

Dart is a programming language developed by Google to tackle the former option by optimizing the use of in-device computation. It can do so because it can be compiled to the ARM machine code used in mobile devices. Dart is used to develop desktop, mobile, web, and server applications, but in this particular paper we examine Dart in the context of mobile application development, and additionally compare it to current popular languages and paradigms.

I. Dart 2.7 Overview

1.1 Internal Implementation Features

There are 4 primary ways to run Dart code. The first is to compile Dart code to JavaScript, allowing Dart code to be runnable on and compliant with all major browsers. Dart can also be run using the Dart Software Development Kit (SDK), which comes with a Dart Virtual Machine that allows one to run Dart code in a command-line interface. The dart2native compiler also allows Dart to be compiled to native, self-contained executable code. Lastly, and perhaps the most pertinent method of execution is Ahead-of-time (AOT) compiled Dart.

Dart code is AOT-compiled into ARM or X64 machine code that can interface more directly with mobile hardware, such that Dart applications compiled in such a way can start up almost instantly and run smoothly in a Dart runtime.

In spite of being single-threaded, Dart also offers support for concurrency in the form of asynchronous programming. The mechanism that allows for the asynchronicity is the classic Event Loop - however what allows Dart to mimic multithreading is an entity known as an *isolate*. An *isolate* can be thought of as a process without all the overhead - it has its own private memory and its own event loop. *Isolates* cannot share memory with one another, which may be a difficult implementation for C++ programmers to deal with. However, by doing so and “isolating” memory from different tasks, we can ensure thread safety since Dart is a single-threaded language which means that memory isn’t being mutated. However, *isolates* can still communicate with one another through messages they pass.

This single-threaded aspect of Dart comes into play once again for memory allocation and garbage collection - since it is single-threaded, the garbage collector and memory allocator know that memory isn’t being mutated, allowing them to perform their tasks optimally.

Dart’s garbage collector itself is another distinction it offers - it’s an implementation of a generational garbage collector that can offer increased frequencies of object creation and destruction. Since Dart (by means of Flutter) is a language used to manage many UI components, this is an extremely convenient performance boost to have.

Dart's internal implementation is extremely suitable for developing clean, smooth, and complex mobile applications. The thread safety offered by *isolates* and its single-threaded nature allow for optimal process execution, resulting in better performance from its crucial generational garbage collector, while the AOT compilation allows somewhat bulky and computationally heavy algorithms to be sped up by converting them into machine code prior to deployment.

1.2 Language Syntax, Usage and Libraries

Dart does not employ any access modifiers as in Java - however underscores-prepended identifiers are marked as library-private and will not be exported with the rest of the library functions and variables.

Dart employs a mixture of strict and loose type-checking. It is a matter of personal choice whether or not variable types are declared. However, there are some unique features of the Dart typing system. Firstly, it allows for a *dynamic* type that allows the variable marked with the *dynamic* type to store any type of value, which is allowed to change throughout the execution of the program. When types aren't explicitly declared, Dart employs a type inference system - this type inference can be explicitly invoked using the *var* type.

Asynchronous programming is also supported in Dart through the *async* and *await* keywords, along with the internal Event Loop implementation. This feature is key to being able to perform latency-heavy network requests or database transactions without blocking the rest of the *isolate*. This concurrency allows Dart applications to have smooth, seamless user interfaces that are not impacted by heavy background computations. This naturally lends itself to the particular problem we are attempting to solve - a mobile application performing computationally heavy machine learning calculations in-device. Another key part of Dart's asynchronous programming implementation is the notion of a *Future*, or what's known in JavaScript as a *Promise*, representing objects that resolve to actual values (or errors) when awaited.

Like many other popular languages, such as C/C++, the primary entry point of a Dart program is the `main` *isolate*.

II. Language Comparisons

2.1 OCaml

OCaml and Dart are for the most part quite different languages. Dart is more oriented towards UI-development and application development, whereas OCaml is much more performant on heavy computation. However, both employ an Object Oriented Programming paradigm, and to an extent both implement a type-inference system. Where Dart separates itself from OCaml in typing is the existence of the Dart *dynamic* type. Additionally Dart does offer asynchronous programming support, something difficult to do in OCaml.

2.2 Java

Dart and Java do not offer the same levels of access modification - however, they both have the notion of private objects. Additionally, they are both Object Oriented Programming languages, capable of being partially compiled (Java bytecode in the case of Java and ARM Machine code in the case of Dart). Another key difference is the single-threaded nature of Dart in comparison to the multi-threaded Java. Java also offers more flexibility in thread manipulation, whereas Dart *isolates* restrict memory on a per-isolate basis.

2.3 Python

Dart has more flexibility in execution options compared to Python, which is an interpreted language meaning it will oftentimes be slower than equivalent Dart code. Furthermore, Dart's flexible typing system is similar to that of Python's, except that Dart still allows for explicit type-checking, something not as easily achievable for most Python distributions.

However, both Python and Dart are capable of effectively executing asynchronous code - Dart by

means of built in support and Python through means of the *asyncio* library.

III. Analysis

For the GarageGarner implementation to be modified to support edge computing, computation-heavy operations must not be detrimental to the calculations performed to ensure a smooth user interface/experience. In order to do so effectively, asynchronous programming is crucial. The other concern with moving machine learning algorithms from an external server to the device application is storage - the size of the application could potentially be increased drastically as a result of porting the code to the client. Precompiling this code prior to deployment can help to minimize application storage, a feature offered to us by AOT compilation. AOT compilation can additionally reduce the overhead of starting the application itself. Both of the aforementioned features are supported by Dart, and its generational garbage collector allows it to be especially performant for the user interface calculations crucial for a smooth user experience.

Dart also offers the flexibility of multiple compilation options. Not only can it be compiled to ARM and X64, but it can also be compiled to JavaScript. This key feature can allow Dart code to be integrated with pre existing JavaScript code, which dominates the current mobile application environment.

IV. Conclusion

In spite of how young it is, Dart is a promising and robust language that is extremely well-suited for complex mobile application development, while addressing many of the programmatic flaws that current dominant players in the mobile application market (like JavaScript) have. Its syntax is a hybrid between JavaScript and Java, allowing for easy migration for programmers, while offering the syntactic benefits of each language. However, in spite of its robustness and performance capability, due to the prominence of frameworks such as React-Native and Swift, it is difficult to find the same extensive developer community in Dart+Flutter application developments.

V. References

(1) https://dart.dev/platforms
(2) http://radar.oreilly.com/2012/03/what-is-dart.html
(3) https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bffc3e296a6a