Глава 1. История и эволюция языка Java

Создание языка Java

Главной движущей силой стала потребность в **независимом от платформы** (т.е. архитектурно-нейтральном) языке, который можно было бы использовать для построения программного обеспечении, встраиваемого в разнообразные бытовые электронные устройства, такие как микроволновые печи и пульты дистанционного управления.

Второй движущей силой стала Всемирная паутина (она же веб-сеть) т.к. веб-сеть тоже нуждалась в переносимых программах.

Апплет Java

Java апплет — это Java-приложение, которое запускается внутри веб-браузера с помощью Java-плагина. Апплеты были популярными в 1990-х и начале 2000-х годов как способ предоставления интерактивных графических приложений на веб-страницах. Они позволяли разработчикам создавать множество интересных веб-приложений, включая игры, графические редакторы и интерактивные диаграммы.

По существу, апплет позволяет переместить определенную функциональность из серверной стороны на клиентскую сторону.

Напротив, апплет представляет собой динамическую самостоятельно запускающуюся программу. Такая программа является активным агентом на клиентском компьютере, но инициируется сервером

Основные характеристики Java апплетов включают:

* Платформонезависимость: Java апплеты могли выполняться на разных операционных системах и архитектурах благодаря тому, что они компилировались в байт-код Java и выполнялись на виртуальной машине Java (JVM).
* Безопасность: Апплеты были ограничены в своих возможностях для обеспечения безопасности. Они могли выполняться в песочнице (sandbox), что ограничивало их доступ к ресурсам компьютера.
* Графический интерфейс: Java апплеты могли создавать графический пользовательский интерфейс с использованием библиотеки AWT (Abstract Window Toolkit) или Swing.

Связь Java-кода с браузером осуществлялась с помощью Java апплетов. Вот как это происходило:

* Написание Java-кода: Разработчик создавал Java-код, который выполнял определенные задачи и визуализацию интерфейса, если это требовалось.
* Компиляция в байт-код Java: Java-код затем компилировался в байт-код Java с использованием компилятора Java. Байт-код был независим от конкретной аппаратной платформы.
* Создание HTML-страницы: Разработчик создавал HTML-страницу, которая включала тег <applet>. В этом теге указывалось имя класса Java-апплета и другие параметры, такие как размеры окна, параметры безопасности и т. д.

Пример тега <applet>:



* Загрузка и выполнение в браузере: при загрузке веб-страницы браузер обнаруживал тег <applet>, скачивал байт-код Java-апплета и передавал его виртуальной машине Java (JVM), которая выполняла апплет внутри браузера. Апплет мог взаимодействовать с веб-страницей и рендерить графику, как это требовалось.
* Обработка событий и взаимодействие: Java-апплеты могли реагировать на события в браузере, такие как клики мыши или нажатия клавиш, и выполнять соответствующие действия, например, изменение содержимого на веб-странице.

Магия Java: байт-код

Компилятор Java генерирует не исполняемый код, а байт-код. Байт-код представляет собой оптимизированный набор инструкций, предназначенных для выполнения так называемой виртуальной машиной Java (Java Virtual Machine - JYM), которая является частью исполняющей среды Java (Java Runtime Environment - JRE).

Java Virtual Machine (JVM) — это виртуальная машина, которая выполняет байт-код Java, созданный компилятором Java из исходного Java-кода. JVM обеспечивает платформонезависимость Java, позволяя Java-приложениям выполняться на разных операционных системах без изменений в их исходном коде.

Вот как работает JVM:

1. Компиляция в байт-код: Исходный Java-код компилируется в байт-код Java с помощью Java-компилятора (javac). Байт-код представляет собой набор инструкций, которые являются абстрактными и не зависят от конкретной аппаратной платформы.
2. Загрузка классов: JVM загружает байт-код Java в память по мере необходимости. Классы могут быть загружены с локального класспаса (classpath) или из удаленных источников, таких как удаленные серверы или интернет.
3. Верификация: перед выполнением классы проходят процесс верификации, в ходе которого проверяется их правильность и безопасность. Это важный этап, который защищает систему от вредоносного кода.
4. Исполнение: когда класс загружен и верифицирован, JVM начинает выполнение байт-кода Java. Виртуальная машина интерпретирует байт-код или может использовать Just-In-Time (JIT) компиляцию для перевода байт-кода в нативный машинный код для конкретной аппаратной платформы, что может улучшить производительность приложения.
5. Управление памятью: JVM управляет памятью, выделяя и освобождая память для объектов, создаваемых во время выполнения приложения. Это включает в себя управление кучей (heap) для объектов и стеком вызовов (call stack) для управления вызовами методов и локальными переменными.
6. Сборка мусора: JVM автоматически отслеживает объекты, которые больше не используются, и периодически освобождает память, занимаемую этими объектами, с помощью сборщика мусора (garbage collector).
7. Управление исключениями: JVM обрабатывает исключения, которые могут возникнуть во время выполнения приложения, и предоставляет механизмы для их обработки и управления.
8. Многозадачность: JVM обеспечивает поддержку многозадачности, позволяя выполнять несколько потоков (threads) одновременно в рамках одного процесса Java-приложения.
9. Библиотеки Java: JVM включает в себя стандартные библиотеки Java (Java Standard Library), которые предоставляют множество классов и методов для работы с различными аспектами программирования, такими как работа с файлами, сетевое взаимодействие, графика и многое другое.

Just-In-Time (JIT) компиляция — это техника оптимизации выполнения кода виртуальной машины (VM) или интерпретатора, которая используется, чтобы улучшить производительность при выполнении программы. JIT компиляция предназначена для ускорения работы приложения путем преобразования некоторых частей кода, представленного в байт-коде или другом промежуточном формате, в нативный машинный код, который выполняется непосредственно на целевой аппаратной платформе.

Вот как работает JIT компиляция:

1. Интерпретация и компиляция: изначально, при запуске программы, код выполняется в интерпретируемом режиме. Это означает, что инструкции байт-кода (или другого промежуточного кода) интерпретируются виртуальной машиной, что может быть медленным в сравнении с нативным машинным кодом.
2. Профилирование кода: во время интерпретации виртуальная машина отслеживает, какие части кода выполняются наиболее часто. Это называется профилированием кода.
3. Компиляция в нативный код: когда определенный фрагмент кода выполняется достаточно часто, JIT-компилятор анализирует этот код и транслирует его в нативный машинный код для конкретной аппаратной платформы. Это происходит во время выполнения программы, и только для тех участков кода, которые действительно нуждаются в оптимизации.
4. Кеширование: Сгенерированный нативный код кешируется, чтобы он мог быть использован при последующих вызовах того же фрагмента кода без повторной компиляции.

Нативный машинный код — это низкоуровневый исполняемый код, который представляет собой инструкции, понятные конкретному процессору или архитектуре компьютера. Этот код является непосредственно понятным и выполняемым процессором, без дополнительной обработки или интерпретации.

Байт-код представляет собой абстрактную форму кода, которая является промежуточным уровнем между исходным кодом и нативным машинным кодом. Он не зависит от конкретной аппаратной платформы, так как он представлен в виде набора инструкций и операций, которые понимает виртуальная машина (например, JVM для Java или CLR для C#). Виртуальная машина интерпретирует байт-код или компилирует его в нативный машинный код во время выполнения, адаптируя его под текущую платформу. Таким образом, байт-код обеспечивает переносимость между разными архитектурами и операционными системами.

Как все взаимосвязано:

1. Байт-код Java: когда вы компилируете исходный Java-код с помощью компилятора Java, он создает байт-код Java. Этот байт-код представляет собой инструкции, которые не зависят от конкретной аппаратной платформы, и это делает его переносимым.
2. JVM — это программное обеспечение, которое является частью среды выполнения Java (Java Runtime Environment, JRE). JVM интерпретирует или JIT-компилирует байт-код Java в нативный машинный код, который может выполняться на конкретной аппаратной платформе. Это ключевой момент, который обеспечивает переносимость Java-приложений.
3. Изоляция нативного кода: Нативный код, который генерирует JVM, ассоциирован с аппаратной платформой и операционной системой, на которой JVM работает. Однако этот нативный код находится внутри JVM и изолирован от самой Java-программы. Java-приложение взаимодействует с нативным кодом через API, которые предоставляются JVM, и эти API переносимы и абстрагируют код Java от деталей аппаратной платформы.
4. Платформонезависимость: поскольку байт-код Java является переносимым и может быть выполнен на любой платформе, на которой установлена совместимая JVM, Java-приложения могут быть написаны один раз и выполняться на разных операционных системах и аппаратных платформах, где существует JVM. Это обеспечивает платформонезависимость Java.

API (Application Programming Interface) — это набор определенных правил и протоколов, которые позволяют разным программным компонентам взаимодействовать друг с другом. API определяет, какие функции и методы доступны для использования внешними приложениями, и как эти приложения могут взаимодействовать с системой, библиотекой, сервисом или операционной системой.

Примерами API могут быть следующие:

* API операционной системы, которое позволяет приложениям управлять файлами, сетевыми соединениями и другими аспектами ОС.
* API социальных сетей, которые позволяют разработчикам создавать приложения, интегрированные с социальными платформами.
* API библиотеки, предоставляющей математические функции для приложения.
* API веб-сервисов, позволяющие приложениям обмениваться данными через интернет.

Взаимодействие Java-приложения с нативным кодом происходит через Java Native Interface (JNI), который представляет собой API, предоставляющее возможность вызывать функции, написанные на нативных языках (как правило, на C или C++), из Java-приложения. Это позволяет Java-приложениям использовать библиотеки и функциональность, которая не доступна непосредственно на языке Java.

Сервлеты: Java на серверный стороне

Сервлеты представляют собой Java-классы, которые используются для разработки динамических веб-приложений на платформе Java. Они работают на серверной стороне и предназначены для обработки HTTP-запросов и генерации HTTP-ответов. Сервлеты широко используются в веб-приложениях для создания веб-страниц, взаимодействия с базами данных, обработки форм, аутентификации и других веб-связанных задач.

Вот основные характеристики сервлетов:

1. Java-классы: Сервлеты представляют собой обычные классы на языке Java, которые расширяют классы или реализуют интерфейсы из Java Servlet API. Обычно, для создания сервлета, нужно создать Java-класс и переопределить методы, такие как doGet() или doPost(), для обработки HTTP-запросов.
2. Жизненный цикл: у сервлетов есть жизненный цикл, который включает в себя инициализацию, обработку запросов и завершение работы. Сервлет может быть создан один раз и использоваться для обслуживания нескольких запросов, что позволяет сэкономить ресурсы сервера.
3. Обработка HTTP-запросов: Главная задача сервлетов - обработка HTTP-запросов от клиентов (например, браузеров). Они могут анализировать параметры запроса, читать и записывать HTTP-заголовки, работать с сессиями пользователей и так далее.
4. Генерация HTTP-ответов: Сервлеты также генерируют HTTP-ответы, которые включают в себя HTML-страницы, JSON-данные, изображения и другие ресурсы. Ответы отправляются клиентам (браузерам) в форме HTTP-сообщений.
5. Поддержка многопоточности: Сервлеты автоматически поддерживают многопоточность. Они могут одновременно обслуживать несколько запросов от разных клиентов, и каждый запрос выполняется в отдельном потоке.
6. Конфигурация через web.xml или аннотации: Сервлеты могут быть настроены с использованием файлов конфигурации web.xml или с помощью аннотаций (в более современных версиях Java Servlet API). Это позволяет определить URL-пути, по которым сервлеты будут доступны, а также другие параметры.

Сервлеты являются важной частью технологии Java EE (Java Platform, Enterprise Edition) и используются для создания множества веб-приложений, включая корпоративные приложения, интернет-магазины, порталы и другие веб-ресурсы. Они предоставляют программистам мощный инструмент для создания динамических и интерактивных веб-приложений, работающих на сервере Java.

Терминология языка Java

• простота; • безопасность; • переносимость; • объектная ориентация; • надежность; • многопоточность; • нейтральность к архитектуре; • интерпретируемость; • высокая производительность; • распределенность; • динамичность.

*Надежность.*

Java – строго типизированный язык.

Строго типизированный язык (или сильно типизированный язык) — это язык программирования, который строго следует правилам и ограничениям, касающимся типов данных. В строго типизированных языках каждая переменная и выражение имеют строго определенный тип данных, и их использование ограничено в соответствии с этим типом.

Чтобы лучше понять, как обеспечивается надежность в Java, рассмотрим две главных причины отказа программ: просчеты в управлении памятью и неправильно обработанные исключительные ситуации (т.е. ошибки времени выполнения).

Скажем, в С/С++ программист будет часто вручную выделять и освобождать динамическую память. Подход подобного рода иногда приводит к возникновению проблем, потому что программисты будут либо забывать об освобождении ранее выделенной памяти, либо, что хуже, пытаться освободить память, которая все еще задействована в другой части кода. Java практически устраняет указанные проблемы, самостоятельно управляя выделением и освобождением памяти. (На самом деле освобождение выполняется полностью автоматически, поскольку Java обеспечивает сборку мусора для неиспользуемых объектов.)

Условия для исключений в традиционных средах часто возникают в ситуациях вроде деления на ноль или отсутствия нужного файла, и справляться с ними приходится с помощью неуклюжих и трудных для чтения конструкций. Java помогает и этой области, предлагая объектно-ориентированную обработку исключений. В хорошо написанной программе на Java все ошибки времени выполнения могут - и должны - обрабатываться вашей программой.

*Интерпретируемость и высокая производительность*

Байт-код Java был тщательно спроектирован, чтобы легко транслироваться прямо в машинный код для достижения очень высокой производительности с использованием оперативного компилятора.