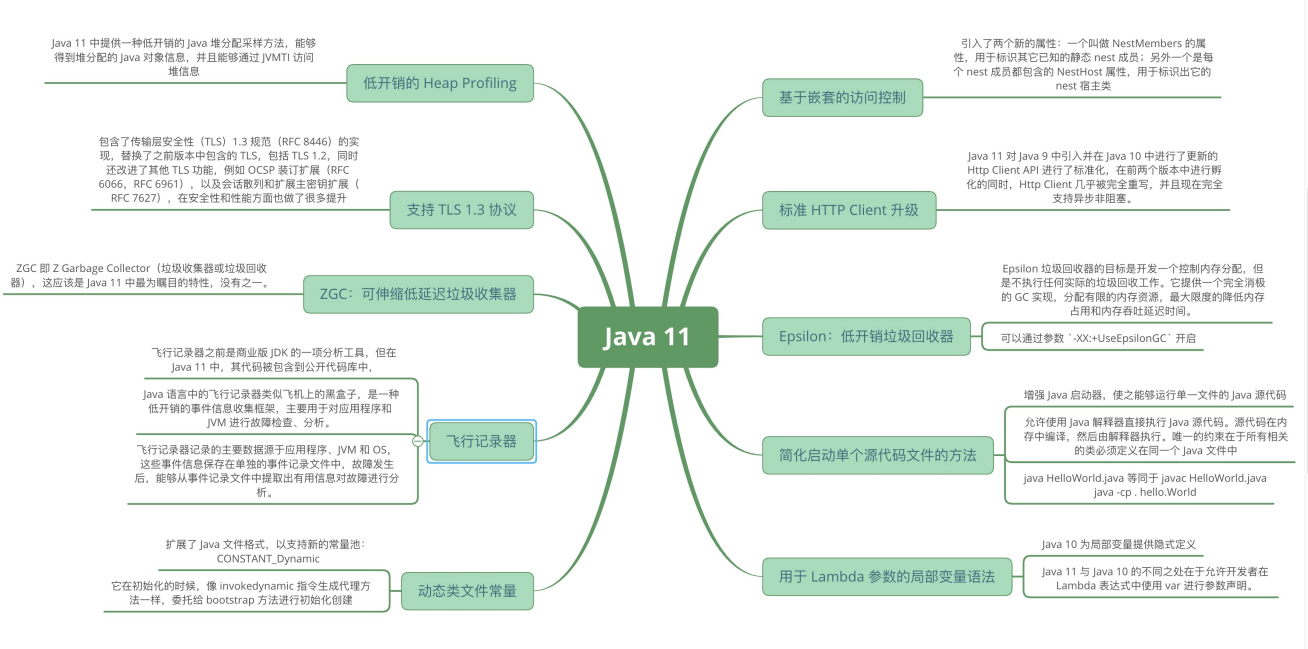
Java 11 新特性概述

知识体系



Java 11 已于 2018 年 9 月 25 日正式发布，之前在 Java 10 新特性介绍 中介绍过，为了加快的版本迭代、跟进社区反馈，Java 的版本发布周期调整为每六个月一次——即每半年发布一个大版本，每个季度发布一个中间特性版本，并且做出不会跳票的承诺。通过这样的方式，Java 开发团队能够将一些重要特性尽早的合并到 Java Release 版本中，以便快速得到开发者的反馈，避免出现类似 Java 9 发布时的两次延期的情况。

按照官方介绍，新的版本发布周期将会严格按照时间节点，于每年的 3 月和 9 月发布，Java 11 发布的时间节点也正好处于 Java 8 免费更新到期的前夕。与 Java 9 和 Java 10 这两个被称为”功能性的版本”不同，Java 11 仅将提供长期支持服务（LTS, Long-Term-Support），还将作为 Java 平台的默认支持版本，并且会提供技术支持直至 2023 年 9 月，对应的补丁和安全警告等支持将持续至 2026 年。

基于嵌套的访问控制

与 Java 语言中现有的嵌套类型概念一致, 嵌套访问控制是一种控制上下文访问的策略，允许逻辑上属于同一代码实体，但被编译之后分为多个分散的 class 文件的类，无需编译器额外的创建可扩展的桥接访问方法，即可访问彼此的私有成员，并且这种改进是在 Java 字节码级别的。

在 Java 11 之前的版本中，编译之后的 class 文件中通过 InnerClasses 和 Enclosing Method 两种属性来帮助编译器确认源码的嵌套关系，每一个嵌套的类会编译到自己所在的 class 文件中，不同类的文件通过上面介绍的两种属性的来相互连接。这两种属性对于编译器确定相互之间的嵌套关系已经足够了，但是并不适用于访问控制。这里大家可以写一段包含内部类的代码，并将其编译成 class 文件，然后通过 javap 命令行来分析，碍于篇幅，这里就不展开讨论了。

Java 11 中引入了两个新的属性：一个叫做 NestMembers 的属性，用于标识其它已知的静态 nest 成员；另外一个是每个 nest 成员都包含的 NestHost 属性，用于标识出它的 nest 宿主类。

标准 HTTP Client 升级

Java 11 对 Java 9 中引入并在 Java 10 中进行了更新的 Http Client API 进行了标准化，在前两个版本中进行孵化的同时，Http Client 几乎被完全重写，并且现在完全支持异步非阻塞。

新版 Java 中，Http Client 的包名由 jdk.incubator.http 改为 java.net.http，该 API 通过 CompleteableFutures 提供非阻塞请求和响应语义，可以联合使用以触发相应的动作，并且 RX Flo w 的概念也在 Java 11 中得到了实现。现在，在用户层请求发布者和响应发布者与底层套接字之间追踪数据流更容易了。这降低了复杂性，并最大程度上提高了 HTTP/1 和 HTTP/2 之间的重用的可能性。

Java 11 中的新 Http Client API，提供了对 HTTP/2 等业界前沿标准的支持，同时也向下兼容 HTTP/1.1，精简而又友好的 API 接口，与主流开源 API（如：Apache HttpClient、Jetty、OkHttp 等）类似甚至拥有更高的性能。与此同时它是 Java 在 Reactive-Stream 方面的第一个生产实践，其中广泛使用了 Java Flow API，终于让 Java 标准 HTTP 类库在扩展能力等方面，满足了现代互联网的需求，是一个难得的现代 Http/2 Client API 标准的实现，Java 工程师终于可以摆脱老旧的 HttpURLConnection 了。下面模拟 Http GET 请求并打印返回内容：

清单 1. GET 请求示例

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create("http://openjdk.java.net/"))
      .build();
client.sendAsync(request, BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();
```

Epsilon：低开销垃圾回收器

Epsilon 垃圾回收器的目标是开发一个控制内存分配，但是不执行任何实际的垃圾回收工作。它提供一个完全消极的 GC 实现，分配有限的内存资源，最大限度的降低内存占用和内存吞吐延迟时间。

Java 版本中已经包含了一系列的高度可配置化的 GC 实现。各种不同的垃圾回收器可以面对各种情况。但是有些时候使用一种独特的实现，而不是将其堆积在其他 GC 实现上将会是事情变得更加简单。

下面是 no-op GC 的几个使用场景：

- **性能测试**：什么都不执行的 GC 非常适合用于 GC 的差异性分析。no-op （无操作）GC 可以用于过滤掉 GC 诱发的性能损耗，比如 GC 线程的调度，GC 屏障的消耗，GC 周期的不合适触发，内存位置变化等。此外有些延迟者不是由于 GC 引起的，比如 scheduling hiccups, compiler transition hiccups，所以去除 GC 引发的延迟有助于统计这些延迟。
- **内存压力测试**：在测试 Java 代码时，确定分配内存的阈值有助于设置内存压力常量值。这时 no-op 就很有用，它可以简单地接受一个分配的内存分配上限，当内存超限时就失败。例如：测试需要分配小于 1G 的内存，就使用-Xmx1g 参数来配置 no-op GC，然后当内存耗尽的时候就直接 crash。
- **VM 接口测试**：以 VM 开发视角，有一个简单的 GC 实现，有助于理解 VM-GC 的最小接口实现。它也用于证明 VM-GC 接口的健全性。
- **极度短暂 job 任务**：一个短声明周期的 job 任务可能会依赖快速退出来释放资源，这个时候接收 GC 周期来清理 heap 其实是在浪费时间，因为 heap 会在退出时清理。并且 GC 周期可能会占用一会时间，因为它依赖 heap 上的数据量。 延迟改进：对那些极端延迟敏感的应用，开发者十分清楚内存占用，或者是几乎没有垃圾回收的应用，此时耗时较长的 GC 周期将会是一件坏事。

- **吞吐改进**：即便对那些无需内存分配的工作，选择一个 GC 意味着选择了一系列的 GC 屏障，所有的 OpenJDK GC 都是分代的，所以他们至少会有一个写屏障。避免这些屏障可以带来一点点的吞吐量提升。

Epsilon 垃圾回收器和其他 OpenJDK 的垃圾回收器一样，可以通过参数 `-XX:+UseEpsilonGC` 开启。

Epsilon 线性分配单个连续内存块。可复用现存 VM 代码中的 TLAB 部分的分配功能。非 TLAB 分配也是同一段代码，因为在此方案中，分配 TLAB 和分配大对象只有一点点的不同。Epsilon 用到的 barrier 是空的(或者说是无操作的)。因为该 GC

执行任何的 GC 周期，不用关系对象图，对象标记，对象复制等。引进一种新的 barrier-set 实现可能是该 GC 对 JVM 最大的变化。

简化启动单个源代码文件的方法

Java 11 版本中最令人兴奋的功能之一是增强 Java 启动器，使之能够运行单一文件的 Java 源代码。此功能允许使用 Java 解释器直接执行 Java 源代码。源代码在内存中编译，然后由解释器执行。唯一的约束在于所有相关的类必须定义在同一个 Java 文件中。

此功能对于开始学习 Java 并希望尝试简单程序的人特别有用，并且能与 jshell 一起使用，将成为任何初学者学习语言的一个很好的工具集。不仅初学者会受益，专业人员还可以利用这些工具来探索新的语言更改或尝试未知的 API。

如今单文件程序在编写小实用程序时很常见，特别是脚本语言领域。从中开发者可以省去用 Java 编译程序等不必要工作，以及减少新手的入门障碍。在基于 Java 10 的程序实现中可以通过三种方式启动：

- 作为 * .class 文件
- 作为 * .jar 文件中的主类
- 作为模块中的主类

而在最新的 Java 11 中新增了一个启动方式，即可以在源代码中声明类，例如：如果名为 HelloWorld.java 的文件包含一个名为 hello.World 的类，那么该命令：

```
$ java HelloWorld.java
```

也等同于：

```
$ javac HelloWorld.java
$ java -cp . hello.World
```

用于 Lambda 参数的局部变量语法

在 Lambda 表达式中使用局部变量类型推断是 Java 11 引入的唯一与语言相关的特性，这一节，我们将探索这一新特性。

从 Java 10 开始，便引入了局部变量类型推断这一关键特性。类型推断允许使用关键字 var 作为局部变量的类型而不是实际类型，编译器根据分配给变量的值推断出类型。这一改进简化了代码编写、节省了开发者的工作时间，因为不再需要显式声明局部变量的类型，而是可以使用关键字 var，且不会使源代码过于复杂。

可以使用关键字 var 声明局部变量，如下所示：

```
var s = "Hello Java 11";
System.out.println(s);
```

但是在 Java 10 中，还有下面几个限制：

- 只能用于局部变量上
- 声明时必须初始化
- 不能用作方法参数
- 不能在 Lambda 表达式中使用

Java 11 与 Java 10 的不同之处在于允许开发者在 Lambda 表达式中使用 var 进行参数声明。乍一看，这一举措似乎有点多余，因为在写代码过程中可以省略 Lambda 参数的类型，并通过类型推断确定它们。但是，添加上类型定义同时使用 @Nonnull 和 @Nullable 等类型注释还是很有用的，既能保持与局部变量的一致写法，也不丢失代码简洁。

Lambda 表达式使用隐式类型定义，它形参的所有类型全部靠推断出来的。隐式类型 Lambda 表达式如下：

```
(x, y) -> x.process(y)
```

Java 10 为局部变量提供隐式定义写法如下：

```
var x = new Foo();
for (var x : xs) { ... }
try (var x = ...) { ... } catch ...
```

为了 Lambda 类型表达式中正式参数定义的语法与局部变量定义语法的不一致，且为了保持与其他局部变量用法上的一致性，希望能够使用关键字 var 隐式定义 Lambda 表达式的形参：

```
(var x, var y) -> x.process(y)
```

于是在 Java 11 中将局部变量和 Lambda 表达式的用法进行了统一，并且可以将注释应用于局部变量和 Lambda 表达式：

```
@Nonnull var x = new Foo();
(@Nonnull var x, @Nullable var y) -> x.process(y)
```

低开销的 Heap Profiling

Java 11 中提供一种低开销的 Java 堆分配采样方法，能够得到堆分配的 Java 对象信息，并且能够通过 JVMTI 访问堆信息。

引入这个低开销内存分析工具是为了达到如下目的：

- 足够低的开销，可以默认且一直开启
- 能通过定义好的程序接口访问
- 能够对所有堆分配区域进行采样
- 能给出正在和未被使用的 Java 对象信息

对用户来说，了解它们堆里的内存分布是非常重要的，特别是遇到生产环境中出现的高 CPU、高内存占用率的情况。目前有一些已经开源的工具，允许用户分析应用程序中的堆使用情况，比如：Java Flight Recorder、jmap、YourKit 以及 VisualVM tools.。但是这些工具都有一个明显的不足之处：无法得到对象的分配位置，headp dump 以及 heap histogram 中都没有包含对象分配的具体信息，但是这些信息对于调试内存问题至关重要，因为它能够告诉开发人员他们的代码中发生的高内存分配的确切位置，并根据实际源码来分析具体问题，这也是 Java 11 中引入这种低开销堆分配采样方法的原因。

支持 TLS 1.3 协议

Java 11 中包含了传输层安全性（TLS）1.3 规范（RFC 8446）的实现，替换了之前版本中包含的 TLS，包括 TLS 1.2，同时还改进了其他 TLS 功能，例如 OCSP 装订扩展（RFC 6066，RFC 6961），以及会话散列和扩展主密钥扩展（RFC 7627），在安全性和性能方面也做了很多提升。

新版本中包含了 Java 安全套接字扩展（JSSE）提供 SSL，TLS 和 DTLS 协议的框架和 Java 实现。目前，JSSE API 和 JDK 实现支持 SSL 3.0，TLS 1.0，TLS 1.1，TLS 1.2，DTLS 1.0 和 DTLS 1.2。

同时 Java 11 版本中实现的 TLS 1.3，重新定义了以下新标准算法名称：

- TLS 协议版本名称：TLSv1.3
- SSLContext 算法名称：TLSv1.3
- TLS 1.3 的 TLS 密码套件名称：TLS_AES_128_GCM_SHA256，TLS_AES_256_GCM_SHA384
- 用于 X509KeyManager 的 keyType：RSASSA-PSS
- 用于 X509TrustManager 的 authType：RSASSA-PSS

还为 TLS 1.3 添加了一个新的安全属性 jdk.tls.keyLimits。当处理了特定算法的指定数据量时，触发握手后，密钥和 IV 更新以导出新密钥。还添加了一个新的系统属性 jdk.tls.server.protocols，用于在 SunJSSE 提供程序的服务器端配置默认启用的协议套件。

之前版本中使用的 KRB5 密码套件实现已从 Java 11 中删除，因为该算法已不再安全。同时注意，TLS 1.3 与以前的版本不直接兼容。

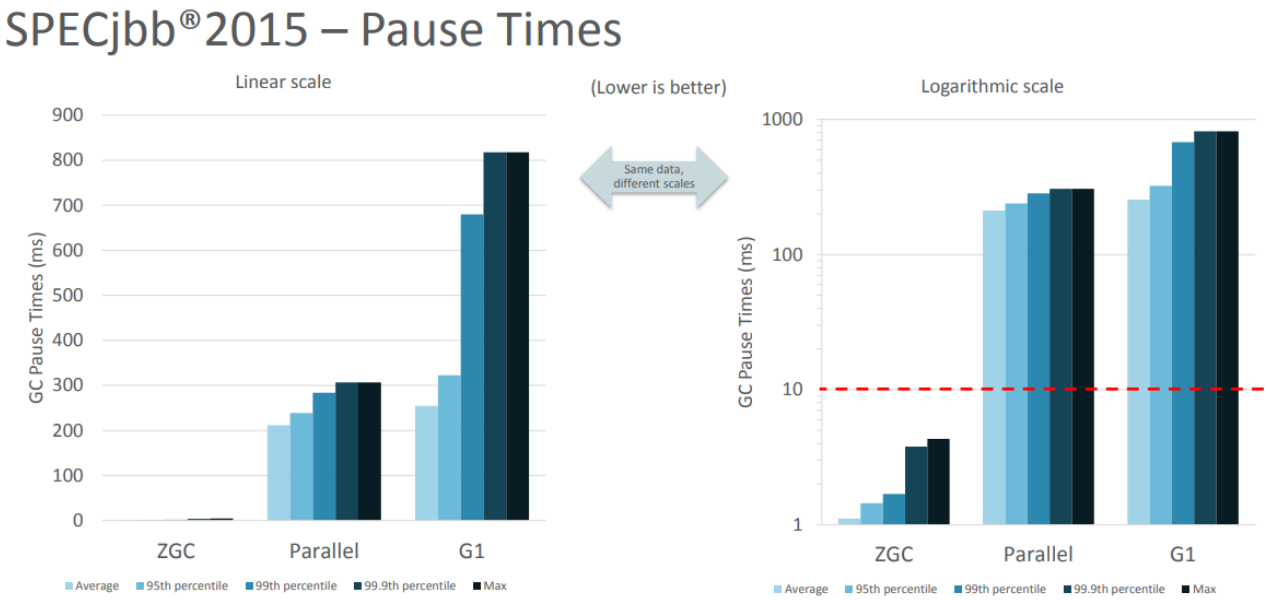
升级到 TLS 1.3 之前，需要考虑如下几个兼容性问题：

- TLS 1.3 使用半关闭策略，而 TLS 1.2 以及之前版本使用双工关闭策略，对于依赖于双工关闭策略的应用程序，升级到 TLS 1.3 时可能存在兼容性问题。
- TLS 1.3 使用预定义的签名算法进行证书身份验证，但实际场景中应用程序可能会使用不被支持的签名算法。
- TLS 1.3 再支持 DSA 签名算法，如果在服务器端配置为仅使用 DSA 证书，则无法升级到 TLS 1.3。
- TLS 1.3 支持的加密套件与 TLS 1.2 和早期版本不同，若应用程序硬编码了加密算法单元，则在升级的过程中需要修改相应代码才能升级使用 TLS 1.3。
- TLS 1.3 版本的 session 用行为及秘钥更新行为与 1.2 及之前的版本不同，若应用依赖于 TLS 协议的握手过程细节，则需要注意。

ZGC：可伸缩低延迟垃圾收集器

ZGC 即 Z Garbage Collector（垃圾收集器或垃圾回收器），这应该是 Java 11 中最为瞩目的特性，没有之一。ZGC 是一个可伸缩的、低延迟的垃圾收集器，主要为了满足如下目标进行设计：

- GC 停顿时间不超过 10ms
- 即能处理几百 MB 的小堆，也能处理几个 TB 的大堆
- 应用吞吐能力不会下降超过 15%（与 G1 回收算法相比）
- 方便在此基础上引入新的 GC 特性和利用 colord
- 针以及 Load barriers 优化奠定基础
- 当前只支持 Linux/x64 位平台 停顿时间在 10ms 以下，10ms 其实是一个很保守的数据，即便是 10ms 这个数据，也是 GC 调优几乎达不到的极值。根据 SPECjbb 2015 的基准测试，128G 的大堆下最大停顿时间才 1.68ms，远低于 10ms，和 G1 算法相比，改进非常明显。



本图片引用自： The Z Garbage Collector – An Introduction

不过目前 ZGC 还处于实验阶段，目前只在 Linux/x64 上可用，如果有足够的需求，将来可能会增加对其他平台的支持。同时作为实验性功能的 ZGC 将不会出现在 JDK 构建中，除非在编译时使用 configure 参数：`--with-jvm-features=zgc` 显式启用。

在实验阶段，编译完成之后，已经迫不及待的想试试 ZGC，需要配置以下 JVM 参数，才能使用 ZGC，具体启动 ZGC 参数如下：

```
-XX：+ UnlockExperimentalVMOptions -XX：+ UseZGC -Xmx10g
```

其中参数： -Xmx 是 ZGC 收集器中最重要的调优选项，大大解决了程序员在 JVM 参数调优上的困扰。ZGC 是一个并发收集器，必须要设置一个最大堆的大小，应用需要多大的堆，主要有下面几个考量：

- 对象的分配速率，要保证在 GC 的时候，堆中有足够的内存分配新对象。
- 一般来说，给 ZGC 的内存越多越好，但是也不能浪费内存，所以要找到一个平衡。

飞行记录器

飞行记录器之前是商业版 JDK 的一项分析工具，但在 Java 11 中，其代码被包含到公开代码库中，这样所有人都能使用该功能了。

Java 语言中的飞行记录器类似飞机上的黑盒子，是一种低开销的事件信息收集框架，主要用于对应用程序和 JVM 进行故障检查、分析。飞行记录器记录的主要数据源于应用程序、JVM 和 OS，这些事件信息保存在单独的事件记录文件中，故障发生后，能够从事件记录文件中提取出有用信息对故障进行分析。

启用飞行记录器参数如下：

```
-XX:StartFlightRecording
```

也可以使用 bin/jcmd 工具启动和配置飞行记录器：

清单 2. 飞行记录器启动、配置参数示例

```
$ jcmd <pid> JFR.start
$ jcmd <pid> JFR.dump filename=recording.jfr
$ jcmd <pid> JFR.stop
```

JFR 使用测试：

清单 3. JFR 使用示例

```
public class FlightRecorderTest extends Event {
    @Label("Hello World")
    @Description("Helps the programmer getting started")
    static class HelloWorld extends Event {
        @Label("Message")
        String message;
    }

    public static void main(String[] args) {
        HelloWorld event = new HelloWorld();
        event.message = "hello, world!";
        event.commit();
    }
}
```

在运行时加上如下参数：

```
java -XX:StartFlightRecording=duration=1s, filename=recording.jfr
```

下面读取上一步中生成的 JFR 文件：recording.jfr

清单 4. 飞行记录器分析示例

```
public void readRecordFile() throws IOException {
    final Path path = Paths.get("D:\\ java \\recording.jfr");
    final List<RecordedEvent> recordedEvents = RecordingFile.readAllEvents(path);
    for (RecordedEvent event : recordedEvents) {
        System.out.println(event.getStartTime() + "," + event.getValue("message"));
    }
}
```

动态类文件常量

为了使 JVM 对动态语言更具吸引力，Java 的第七个版本已将 invokedynamic 引入其指令集。

不过 Java 开发人员通常不会注意到此功能，因为它隐藏在 Java 字节代码中。通过使用 invokedynamic，可以延迟方法调用的绑定，直到第一次调用。例如，Java 语言使用该技术来实现 Lambda 表达式，这些表达式仅在首次使用时才显示出来。这样做，invokedynamic 已经演变成一种必不可少的语言功能。

Java 11 引入了类似的机制，扩展了 Java 文件格式，以支持新的常量池：CONSTANT_Dynamic，它在初始化的时候，像 invokedynamic 指令生成代理方法一样，委托给 bootstrap 方法进行初始化创建，对上层软件没有很大的影响，降低开发新形式的可实现类文件约束带来的成本和干扰。

结束语

Java 在更新发布周期为每半年发布一次之后，在合并关键特性、快速得到开发者反馈等方面，做得越来越好。