你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？

你好，我是winter。今天我们的主题是工具。

古语云：“工欲善其事，必先利其器”，程序员群体对工具的爱好和重视是一个悠久的传统。简单趁手的工具是程序员开发的好帮手。

但是在工程方面，工具不仅仅是简单的“趁手”即可，假如一个团队人人都自己发明几个小工具，那么后果将会是灾难性的：同一个团队的同学无法互相配合写代码，一旦有人离职，可能某一个项目就永远无法跑起来了。

所以我们今天从工程的角度谈一谈工具体系的规划。

工具总论

跟性能不同，工具体系并非业务结果，所以我们没法用简单的数据指标来衡量工具，它的结果更多程度是一种开发体验：帮助技术团队内的同学提升效率和体验。

作为工程体系，我们考虑工具的时候同样要遵循基本规则：现状与指标、方案、实施、结果和监控。

不过，对工具而言，指标和结果都是一种“软性指标”，也就是团队的开发效率和开发体验。这里我不太推荐把开发效率和开发体验过度数据化，我的经验是：开发效率提升n倍永远是一种臆想或者主观论断。

工具体系的目标

前面已经讲到，工具是为技术团队本身服务的工程体系，那么，工具的目标是什么呢？其实每一种工具的出现，必然都有一个非常具体的目标，比如npm帮助我们进行包管理，Yeoman帮助我们初始化项目模板。

但是这些目标是工具的目标，不是工具体系的目标。我们做一个假设，**假如你是一个前端团队的工具体系负责人，现在要你来规划团队的工具体系，你会怎么做呢？**

如果你到社区找了一大堆工具，并且把它们要解决的问题都罗列出来，作为工具体系的目标，那就完全走上了错误的道路。

实际上，在考虑具体的工具之前，我们应该解决工具体系的“元问题”，即：我们对工具本身的要求是什么？

考虑到工程行为都是团队合作，我们对工具最基本的要求就是：版本一致。

只有整个团队的工具版本一致，至少要做到避免大版本差异，才能做到互相接手代码时，团队成员能够正确的使用工具开发。

工具体系的另一个重要需求是：避免冲突，一些工具可能互相没有干扰，比如Yeoman和gulp，有一些工具则由社区设计了配合方案，比如webpack和babel，有一些工具，则存在着根本性冲突，如gulp和grunt。

所以，在谈及具体问题之前，我们必须要有这两个要求的解决方案。**这就需要引入一个新的概念：工具链。**

工具链是一系列互相配合的工具，能够协作完成开发任务（注：工具链这个词最早是由C/C++程序员引入的概念，一般包含编译、链接、调试等工具）。

下面我们就来谈谈工具链的设计。

工具体系的设计

要想设计一个工具链，首先我们需要整理一下，前端开发大约要做哪些事，下面是我的答案：

- 初始化项目；
- 运行和调试；
- 测试（单元测试）；
- 发布。

那么，一个前端项目的工具链，大约就会包含这些功能。一个典型的社区项目工具链可能就类似下面这样：

- Yeoman
- webpack
- ava/nyc
- aws-cli

但是，这显然不够，我们还需要一种机制，保证团队使用的工具版本一致。

轻量级的做法是，在项目初始化模板中定义npm script并且在npm dev-dependency中规定它的版本号。

重量级的做法是，开发一个包装工具，在命令行中不直接使用命令，而使用包装过的命令。如在我之前的团队，使用的工具名为def，它规定了一些命令：

- def init
- def dev
- def test
- def publish

这样，工具链的使用者只需指定工具链名称，就不需要知道项目具体使用了哪些工具，这样只需要专注自己的需求就够了。

同时，统一的命令行入口，意味着整个团队不需要互相学习工具链，就可以接手别人的项目开发。

在稍微大一些的团队内部，往往会需要不止一种开发模式，如移动开发和桌面开发，这样，所需要的工具链也不一样，因此我们需要多条工具链。

要想开发新的工具链，可以使用复制分支的方式来扩展原来的工具链。在我原来的工作中，不同的工具链被称作“套件”，每一种套件对应着一组互相配合的工具。

工具体系的执行

因为工具体系服务的是团队内部成员，所以执行非常简单，同时，工具体系的入口是初始化项目，所以只要初始化工具在手，可以控制其它所有工具。

我们在性能的那一课里，已经讲过工程体系的执行分成三个层次：纯管理、制度化和自动化。

工具体系因为其自身特性，可以说是最容易做到自动化的一个体系了。

工具体系的监控

工具体系的结果虽然是软性的，也不能完全不做监控。

纯粹的社区方案比较难做到监控，但是如果我们使用了前面提到的统一命令行入口包装，那么就可以做一些简单的统计工作了。

一般来说，以下指标跟开发者体验较为相关：

- 调试/构建次数；
- 构建平均时长；
- 使用的工具版本；
- 发布次数。

在我之前的工作中，工具团队曾经从构建平均时长数据中发现构建效率问题，对webpack做了大量深度优化来改善开发体验。

同时，工具的相关数据还能够帮助发现一些问题，比如某个项目频繁发布，可能说明它风险很高。工具的相关数据还能帮我们发现老旧的工具，如果某个套件使用频率极低，则可以考虑把它下线。

总之，工具体系的监控不仅仅是衡量工具体系的好帮手，也是非常珍贵的研发数据，里面有很多可挖掘的价值。

总结

这一课，我们讲解了工具相关的工程知识。

我们仍然从目标、方案设计、执行和结果四个方面来讲解，工具体系的目标除了单个工具解决具体问题之外，还要注意一致性和配合问题，因此我们需要工具链。

工具链一般会涵盖研发阶段的各个主要操作。工具体系的执行比较简单，很容易就可以做到完全的自动化。工具体系的监控同样非常重要，工具的监控除了帮助我们改进工具体系，对研发体系的其它部分也有帮助。

最后，请你思考下自己所在的团队，是否已经建立了工具体系？听完了今天的课程，你认为它有哪些可改进的部分？